

EcoStruxure™ Control Expert

Program Languages and Structure

Reference Manual

Original instructions

35006144.25
02/2022

Legal Information

The Schneider Electric brand and any trademarks of Schneider Electric SE and its subsidiaries referred to in this guide are the property of Schneider Electric SE or its subsidiaries. All other brands may be trademarks of their respective owners.

This guide and its content are protected under applicable copyright laws and furnished for informational use only. No part of this guide may be reproduced or transmitted in any form or by any means (electronic, mechanical, photocopying, recording, or otherwise), for any purpose, without the prior written permission of Schneider Electric.

Schneider Electric does not grant any right or license for commercial use of the guide or its content, except for a non-exclusive and personal license to consult it on an "as is" basis. Schneider Electric products and equipment should be installed, operated, serviced, and maintained only by qualified personnel.

As standards, specifications, and designs change from time to time, information contained in this guide may be subject to change without notice.

To the extent permitted by applicable law, no responsibility or liability is assumed by Schneider Electric and its subsidiaries for any errors or omissions in the informational content of this material or consequences arising out of or resulting from the use of the information contained herein.

As part of a group of responsible, inclusive companies, we are updating our communications that contain non-inclusive terminology. Until we complete this process, however, our content may still contain standardized industry terms that may be deemed inappropriate by our customers.

Table of Contents

| | |
|---|-----------|
| Safety Information | 13 |
| Before You Begin..... | 14 |
| Start-up and Test..... | 15 |
| Operation and Adjustments | 16 |
| About the Book | 17 |
| General Presentation of Control Expert | 19 |
| Presentation | 20 |
| Capabilities of Control Expert | 20 |
| User Interface | 26 |
| Project Browser..... | 27 |
| User Application and Project File Formats | 28 |
| Configurator | 34 |
| Data Editor..... | 38 |
| Program Unit Data Editor | 47 |
| Program Editor..... | 52 |
| Function Block Diagram FBD..... | 55 |
| Ladder Diagram (LD) Language | 57 |
| General Information about SFC Sequence Language..... | 59 |
| Instruction List IL | 62 |
| Structured Text ST..... | 64 |
| PLC Simulator..... | 65 |
| Export/Import | 67 |
| User Documentation | 68 |
| Debug Services..... | 69 |
| Diagnostic Viewer..... | 75 |
| Operator Screen..... | 75 |
| Application Structure | 79 |
| Description of the Available Functions for Each Type of PLC | 80 |
| Functions Available for the Different Types of PLC | 80 |
| Application Program Structure..... | 85 |
| Description of Tasks and Processes..... | 85 |
| Presentation of the Master Task | 85 |

- Presentation of the Fast Task 86
- Presentation of Auxiliary Tasks 87
- Overview of Event Processing 89
- Description of Program Units 90
 - Description of Program Units 90
- Description of Sections and Subroutines 92
 - Description of Sections 92
 - Description of SFC sections 93
 - Description of Subroutines 95
- Mono Task Execution 96
 - Description of the Master Task Cycle 96
 - Mono Task: Cyclic Execution 97
 - Periodic Execution 98
 - Control of Cycle Time 99
 - Execution of Quantum Sections with Remote Inputs/Outputs 100
- Multitasking Execution 102
 - Multitasking Software Structure 102
 - Sequencing of Tasks in a Multitasking Structure 104
 - Task Control 105
 - Assignment of Input/Output Channels to Master, Fast and Auxiliary Tasks 108
 - Management of Event Processing 110
 - Execution of TIMER-type Event Processing 111
 - Input/Output Exchanges in Event Processing 115
 - How to Program Event Processing 116
- Application Memory Structure 119
 - Input Output Data Addressing Methods 119
 - Input Output Data Addressing Methods 119
- Memory Structure of the Premium, Atrium and Modicon M340 PLCs 123
 - Memory Structure of Modicon M340 PLCs 123
 - Memory Structure of Premium and Atrium PLCs 127
 - Detailed Description of the Memory Zones 129
- Memory Structure of Quantum PLCs 130
 - Memory Structure of Quantum PLCs 130
 - Detailed Description of the Memory Zones 133

| | |
|---|------------|
| Operating Modes | 135 |
| Modicon M340 PLCs Operating Modes | 135 |
| Processing of Power Outage and Restoral of Modicon M340 PLCs..... | 135 |
| Processing on Cold Start for Modicon M340 PLCs..... | 137 |
| Processing on Warm Restart for Modicon M340 PLCs | 141 |
| Automatic Start in RUN for Modicon M340 PLCs | 144 |
| Processing of State RAM on STOP Mode for Modicon M340 PLCs..... | 145 |
| Premium, Quantum PLCs Operating Modes | 145 |
| Processing of Power Outage and Restoral for Premium/Quantum PLCs | 145 |
| Processing on Cold Start for Premium/Quantum PLCs | 147 |
| Processing on Warm Restart for Premium/Quantum PLCs..... | 152 |
| Automatic Start in RUN for Premium/Quantum | 155 |
| PLC HALT Mode | 156 |
| PLC HALT Mode..... | 156 |
| Data Description | 158 |
| General Overview of Data..... | 159 |
| General | 159 |
| General Overview of the Data Type Families | 160 |
| Overview of Data Instances..... | 162 |
| Overview of the Data References..... | 163 |
| Syntax Rules for Type\Instance Names | 164 |
| Data Types | 166 |
| Elementary Data Types (EDT) in Binary Format | 166 |
| Overview of Data Types in Binary Format | 166 |
| Boolean Types..... | 168 |
| Integer Types | 174 |
| The Time Type..... | 176 |
| Elementary Data Types (EDT) in BCD Format | 177 |
| Overview of Data Types in BCD Format..... | 177 |
| The Date Type..... | 178 |
| The Time of Day (TOD) Type | 179 |
| The Date and Time (DT) Type..... | 180 |
| Elementary Data Types (EDT) in Real Format..... | 181 |
| Presentation of the Real Data Type | 182 |

| | |
|--|-----|
| Elementary Data Types (EDT) in Character String Format | 187 |
| Overview of Data Types in Character String Format..... | 187 |
| Elementary Data Types (EDT) in Bit String Format | 189 |
| Overview of Data Types in Bit String Format | 190 |
| Bit String Types | 190 |
| Derived Data Types (DDT/IODDT/Device DDT) | 192 |
| Arrays | 192 |
| Structures | 195 |
| Overview of the Derived Data Type family (DDT)..... | 196 |
| DDT: Mapping Rules | 199 |
| Overview of Input/Output Derived Data Types (IODDT)..... | 203 |
| Overview of Device Derived Data Types (Device DDT) | 204 |
| Device DDT Instance Naming Rule | 205 |
| Function Block Data Types (DFB\EFB)..... | 208 |
| Overview of Function Block Data Type Families | 208 |
| Characteristics of Function Block Data Types (EFB\DFB)..... | 210 |
| Characteristics of Elements Belonging to Function Blocks | 212 |
| Generic Data Types (GDT)..... | 214 |
| Overview of Generic Data Types | 214 |
| Data Types Belonging to Sequential Function Charts (SFC)..... | 216 |
| Overview of the Data Types of the Sequential Function Chart Family | 216 |
| Compatibility Between Data Types..... | 218 |
| Compatibility Between Data Types | 218 |
| Reference Data Type Declarations..... | 222 |
| Reference Data Type Declarations | 222 |
| Data Instances..... | 226 |
| Data Type Instances | 226 |
| Data Instance Attributes | 230 |
| Direct Addressing Data Instances | 233 |
| Data References | 241 |
| References to Data Instances by Value | 241 |
| References to Data Instances by Name..... | 243 |
| References to Data Instances by Address | 245 |
| Data Naming Rules..... | 249 |
| Programming Language..... | 252 |

| | |
|---|-----|
| Function Block Language FBD | 253 |
| General Information about the FBD Function Block Language | 253 |
| Elementary Functions, Elementary Function Blocks, Derived Function Blocks and Procedures (FFBs)..... | 255 |
| Subroutine Calls | 265 |
| Control Elements | 266 |
| Link | 267 |
| Text Object..... | 269 |
| Execution Sequence of the FFBs | 269 |
| Change Execution Sequence | 271 |
| Loop Planning | 276 |
| Ladder Diagram (LD)..... | 278 |
| General Information about the LD Ladder Diagram Language..... | 278 |
| Contacts | 281 |
| Coils..... | 282 |
| Elementary Functions, Elementary Function Blocks, Derived Function Blocks and Procedures (FFBs)..... | 283 |
| Control Elements..... | 294 |
| Operate Blocks and Compare Blocks | 295 |
| Links | 297 |
| Text Object..... | 300 |
| Edge Recognition | 301 |
| Execution Sequence and Signal Flow | 309 |
| Loop Planning | 311 |
| Change Execution Sequence | 313 |
| SFC Sequence Language..... | 319 |
| General Information about SFC Sequence Language..... | 319 |
| General Information about SFC Sequence Language..... | 319 |
| Link Rules | 323 |
| Steps and Macro Steps..... | 324 |
| Step..... | 324 |
| Macro Steps and Macro Sections..... | 327 |
| Actions and Action Sections | 332 |
| Action | 332 |
| Action Section | 333 |

| | |
|---|-----|
| Qualifier | 335 |
| Transitions and Transition Sections | 342 |
| Transition | 342 |
| Transition Section | 343 |
| Jump | 345 |
| Jump | 345 |
| Link | 346 |
| Link | 346 |
| Branches and Merges | 347 |
| Alternative Branches and Alternative Joints | 347 |
| Parallel Branch and Parallel Joint | 348 |
| Text Objects | 350 |
| Text Object | 350 |
| Single-Token | 350 |
| Execution Sequence Single-Token | 350 |
| Alternative String | 351 |
| Sequence Jumps and Sequence Loops | 352 |
| Parallel Strings | 355 |
| Asymmetric Parallel String Selection | 358 |
| Multi-Token | 361 |
| Multi-Token Execution Sequence | 361 |
| Alternative String | 362 |
| Parallel Strings | 365 |
| Jump into a Parallel String | 368 |
| Jump out of a Parallel String | 369 |
| Instruction List (IL) | 374 |
| General Information about the IL Instruction List | 374 |
| General Information about the IL Instruction List | 374 |
| Operands | 377 |
| Modifier | 379 |
| Operators | 381 |
| Subroutine Call | 393 |
| Labels and Jumps | 393 |
| Comment | 395 |

| | |
|--|-----|
| Calling Elementary Functions, Elementary Function Blocks, Derived Function Blocks and Procedures | 395 |
| Calling Elementary Functions | 395 |
| Calling Elementary Function Blocks and Derived Function Blocks | 401 |
| Calling Procedures | 412 |
| Structured Text (ST) | 420 |
| General Information about the Structured Text ST | 420 |
| General Information about Structured Text (ST)..... | 420 |
| Operands | 423 |
| Operators..... | 425 |
| Instructions | 429 |
| Instructions..... | 429 |
| Assignment | 430 |
| Select Instruction IF...THEN...END_IF | 432 |
| Select Instruction ELSE | 433 |
| Select Instruction ELSIF...THEN | 434 |
| Select Instruction CASE...OF...END_CASE | 435 |
| Repeat Instruction FOR...TO...BY...DO...END_FOR | 435 |
| Repeat Instruction WHILE...DO...END_WHILE | 438 |
| Repeat Instruction REPEAT...UNTIL...END_REPEAT | 438 |
| Repeat Instruction EXIT | 439 |
| Subroutine Call..... | 440 |
| RETURN..... | 440 |
| Empty Instruction | 441 |
| Labels and Jumps..... | 441 |
| Comment | 442 |
| Calling Elementary Functions, Elementary Function Blocks, Derived Function Blocks and Procedures | 442 |
| Calling Elementary Functions | 442 |
| Call Elementary Function Block and Derived Function Block | 448 |
| Procedures..... | 457 |
| User Function Blocks (DFB) | 463 |
| Overview of User Function Blocks (DFB)..... | 464 |
| Introduction to User Function Blocks | 464 |
| Implementing a DFB Function Block | 466 |

- Description of User Function Blocks (DFB)..... 469
 - Definition of DFB Function Block Internal Data..... 469
 - DFB Parameters..... 471
 - DFB Variables 475
 - DFB Code Section 476
- User Function Blocks (DFB) Instance 478
 - Creation of a DFB Instance 478
 - Execution of a DFB Instance 479
 - Programming Example for a Derived Function Block (DFB) 480
- Use of the DFBs from the Different Programming Languages 484
 - Rules for Using DFBs in a Program..... 484
 - Use of IODDTs in a DFB..... 487
 - Use of a DFB in a Ladder Language Program 490
 - Use of a DFB in a Structured Text Language Program 492
 - Use of a DFB in an Instruction List Program..... 495
 - Use of a DFB in a Program in Function Block Diagram Language..... 498
- User Diagnostics DFB 501
 - Presentation of User Diagnostic DFBs 501
- Implicit Type Conversion in Control Expert 503
 - Control Expert Implicit Type Conversion 503
 - Control Expert Differences from IEC Recommendations 504
- Appendices 507
 - IEC Compliance 508
 - General Information regarding IEC 61131-3..... 508
 - General information about IEC 61131-3 Compliance..... 508
 - IEC Compliance Tables..... 509
 - Common elements..... 509
 - IL language elements 521
 - ST language elements 522
 - Common graphical elements 523
 - LD language elements 524
 - Implementation-dependent parameters 525
 - Error Conditions..... 528
 - Extensions of IEC 61131-3 529
 - Extensions of IEC 61131-3, 2nd Edition 530

| | |
|-------------------------------|-----|
| Textual language syntax..... | 531 |
| Textual Language Syntax | 531 |
| Glossary..... | 533 |
| Index..... | 547 |

Safety Information

Important Information

Read these instructions carefully, and look at the equipment to become familiar with the device before trying to install, operate, service, or maintain it. The following special messages may appear throughout this documentation or on the equipment to warn of potential hazards or to call attention to information that clarifies or simplifies a procedure.



The addition of this symbol to a “Danger” or “Warning” safety label indicates that an electrical hazard exists which will result in personal injury if the instructions are not followed.



This is the safety alert symbol. It is used to alert you to potential personal injury hazards. Obey all safety messages that follow this symbol to avoid possible injury or death.

DANGER

DANGER indicates a hazardous situation which, if not avoided, **will result in** death or serious injury.

WARNING

WARNING indicates a hazardous situation which, if not avoided, **could result in** death or serious injury.

CAUTION

CAUTION indicates a hazardous situation which, if not avoided, **could result in** minor or moderate injury.

NOTICE

NOTICE is used to address practices not related to physical injury.

Please Note

Electrical equipment should be installed, operated, serviced, and maintained only by qualified personnel. No responsibility is assumed by Schneider Electric for any consequences arising out of the use of this material.

A qualified person is one who has skills and knowledge related to the construction and operation of electrical equipment and its installation, and has received safety training to recognize and avoid the hazards involved.

Before You Begin

Do not use this product on machinery lacking effective point-of-operation guarding. Lack of effective point-of-operation guarding on a machine can result in serious injury to the operator of that machine.

| |
|---|
| ▲ WARNING |
| <p>UNGUARDED EQUIPMENT</p> <ul style="list-style-type: none"> • Do not use this software and related automation equipment on equipment which does not have point-of-operation protection. • Do not reach into machinery during operation. <p>Failure to follow these instructions can result in death, serious injury, or equipment damage.</p> |

This automation equipment and related software is used to control a variety of industrial processes. The type or model of automation equipment suitable for each application will vary depending on factors such as the control function required, degree of protection required, production methods, unusual conditions, government regulations, etc. In some applications, more than one processor may be required, as when backup redundancy is needed.

Only you, the user, machine builder or system integrator can be aware of all the conditions and factors present during setup, operation, and maintenance of the machine and, therefore, can determine the automation equipment and the related safeties and interlocks which can be properly used. When selecting automation and control equipment and related software for a particular application, you should refer to the applicable local and national standards and regulations. The National Safety Council's Accident Prevention Manual (nationally recognized in the United States of America) also provides much useful information.

In some applications, such as packaging machinery, additional operator protection such as point-of-operation guarding must be provided. This is necessary if the operator's hands and other parts of the body are free to enter the pinch points or other hazardous areas and

serious injury can occur. Software products alone cannot protect an operator from injury. For this reason the software cannot be substituted for or take the place of point-of-operation protection.

Ensure that appropriate safeties and mechanical/electrical interlocks related to point-of-operation protection have been installed and are operational before placing the equipment into service. All interlocks and safeties related to point-of-operation protection must be coordinated with the related automation equipment and software programming.

NOTE: Coordination of safeties and mechanical/electrical interlocks for point-of-operation protection is outside the scope of the Function Block Library, System User Guide, or other implementation referenced in this documentation.

Start-up and Test

Before using electrical control and automation equipment for regular operation after installation, the system should be given a start-up test by qualified personnel to verify correct operation of the equipment. It is important that arrangements for such a check are made and that enough time is allowed to perform complete and satisfactory testing.

▲ WARNING

EQUIPMENT OPERATION HAZARD

- Verify that all installation and set up procedures have been completed.
- Before operational tests are performed, remove all blocks or other temporary holding means used for shipment from all component devices.
- Remove tools, meters, and debris from equipment.

Failure to follow these instructions can result in death, serious injury, or equipment damage.

Follow all start-up tests recommended in the equipment documentation. Store all equipment documentation for future references.

Software testing must be done in both simulated and real environments.

Verify that the completed system is free from all short circuits and temporary grounds that are not installed according to local regulations (according to the National Electrical Code in the U.S.A, for instance). If high-potential voltage testing is necessary, follow recommendations in equipment documentation to prevent accidental equipment damage.

Before energizing equipment:

- Remove tools, meters, and debris from equipment.
- Close the equipment enclosure door.

- Remove all temporary grounds from incoming power lines.
- Perform all start-up tests recommended by the manufacturer.

Operation and Adjustments

The following precautions are from the NEMA Standards Publication ICS 7.1-1995 (English version prevails):

- Regardless of the care exercised in the design and manufacture of equipment or in the selection and ratings of components, there are hazards that can be encountered if such equipment is improperly operated.
- It is sometimes possible to misadjust the equipment and thus produce unsatisfactory or unsafe operation. Always use the manufacturer's instructions as a guide for functional adjustments. Personnel who have access to these adjustments should be familiar with the equipment manufacturer's instructions and the machinery used with the electrical equipment.
- Only those operational adjustments actually required by the operator should be accessible to the operator. Access to other controls should be restricted to prevent unauthorized changes in operating characteristics.

About the Book

Document Scope

This manual describes the elements necessary for the programming of M340, M580, Momentum, Premium, Atrium and Quantum PLCs using the EcoStruxure Control Expert programming workshop.

Validity Note

This documentation is valid for EcoStruxure™ Control Expert 15.0 or later.

Related Documents

| Title of documentation | Reference number |
|--|---|
| EcoStruxure™ Control Expert, Operating Modes | 33003101 (English), 33003102 (French), 33003103 (German), 33003104 (Spanish), 33003696 (Italian), 33003697 (Chinese) |
| EcoStruxure™ Control Expert, System Bits and Words, Reference Manual | EIO0000002135 (English), EIO0000002136 (French), EIO0000002137 (German), EIO0000002138 (Italian), EIO0000002139 (Spanish), EIO0000002140 (Chinese) |
| Modicon M580, Hardware, Reference Manual | EIO0000001578 (English), EIO0000001579 (French), EIO0000001580 (German), EIO0000001582 (Italian), EIO0000001581 (Spanish), EIO0000001583 (Chinese) |
| Modicon M580, RIO Modules, Installation and Configuration Guide | EIO0000001584 (English), EIO0000001585 (French), EIO0000001586 (German), EIO0000001587 (Italian), EIO0000001588 (Spanish), EIO0000001589 (Chinese), |
| Modicon M340, CANopen, Setup Manual | 35013944 (English), 35013945 (French), 35013946 (German), 35013948 (Italian), 35013947 (Spanish), 35013949 (Chinese) |
| Modicon X80, Discrete Input/Output Modules, User Manual | 35012474 (English), 35012475 (German), 35012476 (French), 35012477 (Spanish), 35012478 (Italian), 35012479 (Chinese) |
| Modicon X80, Analog Input/Output Modules, User Manual | 35011978 (English), 35011979 (German), 35011980 (French), 35011981 (Spanish), 35011982 (Italian), 35011983 (Chinese) |

| Title of documentation | Reference number |
|--|--|
| Quantum using EcoStruxure™ Control Expert, Hardware Reference Manual | 35010529 (English), 35010530 (French), 35010531 (German), 35013975 (Italian), 35010532 (Spanish), 35012184 (Chinese) |
| Quantum using EcoStruxure™ Control Expert, Hot Standby System, User Manual | 35010533 (English), 35010534 (French), 35010535 (German), 35013993 (Italian), 35010536 (Spanish), 35012188 (Chinese) |
| EcoStruxure™ Control Expert, I/O Management, Block Library | 33002531 (English), 33002532 (French), 33002533 (German), 33003684 (Italian), 33002534 (Spanish), 33003685 (Chinese) |
| EcoStruxure™ Control Expert, System, Block Library | 33002539 (English), 33002540 (French), 33002541 (German), 33003688 (Italian), 33002542 (Spanish), 33003689 (Chinese) |
| EcoStruxure™ Control Expert, Diagnostics, Block Library | 33002523 (English), 33002524 (French), 33002525 (German), 33003680 (Italian), 33002526 (Spanish), 33003681 (Chinese) |

You can download these technical publications, the present document and other technical information from our website www.se.com/en/download/.

Product Related Information

| ⚠ WARNING |
|--|
| <p>UNINTENDED EQUIPMENT OPERATION</p> <ul style="list-style-type: none"> • The application of this product requires expertise in the design and programming of control systems. Only persons with such expertise should be allowed to program, install, alter, and apply this product. • Follow all local and national safety codes and standards. <p>Failure to follow these instructions can result in death, serious injury, or equipment damage.</p> |

General Presentation of Control Expert

What's in This Part

Presentation20

Contents of this Part

This part describes the general design and behavior of an application created with Control Expert.

Presentation

What's in This Chapter

| | |
|--|----|
| Capabilities of Control Expert..... | 20 |
| User Interface..... | 26 |
| Project Browser..... | 27 |
| User Application and Project File Formats..... | 28 |
| Configurator..... | 34 |
| Data Editor..... | 38 |
| Program Unit Data Editor..... | 47 |
| Program Editor..... | 52 |
| Function Block Diagram FBD..... | 55 |
| Ladder Diagram (LD) Language..... | 57 |
| General Information about SFC Sequence Language..... | 59 |
| Instruction List IL..... | 62 |
| Structured Text ST..... | 64 |
| PLC Simulator..... | 65 |
| Export/Import..... | 67 |
| User Documentation..... | 68 |
| Debug Services..... | 69 |
| Diagnostic Viewer..... | 75 |
| Operator Screen..... | 75 |

Overview

This chapter describes the general design and behavior of a project created with Control Expert.

Capabilities of Control Expert

Hardware Platforms

Control Expert supports the following hardware platforms:

- Modicon M340
- Modicon M580
- Quantum
- Momentum
- Premium

- Atrium

Programming Languages

Control Expert provides the following programming languages for creating the user program:

- Function Block Diagram FBD
- Ladder Diagram (LD) language
- Instruction List IL
- Structured Text ST
- Sequential Control SFC
- Ladder Logic 984 (LL984)

All of these programming languages can be used together in the same project.

All these languages (except LL984) conform to IEC 61131-3.

Block Libraries

The blocks that are included in the delivery of Control Expert extensive block libraries extend from blocks for simple boolean operations, through blocks for strings and array operations to blocks for controlling complex control loops.

For a better overview, the different blocks are arranged in libraries, which are then broken down into families.

The blocks can be used in the programming languages FBD, LD, IL, and ST.

Elements of a Program

A program can be constructed from:

- a master task (MAST)
- a FAST task (not available for Momentum)
- one to 4 AUX tasks (not available for Modicon M340 and Momentum)
- Program Units which are assigned one of the defined tasks (available for Modicon M580 and Modicon M340)
- sections, which are assigned one of the defined tasks
- sections for processing time controlled events (Timerx, not available for Momentum)
- sections for processing hardware controlled events (EVTx, not available for Momentum)
- subroutine sections (SR)

Software Licenses

There is one Control Expert installation setup (.iso file) and the license determines the version that can be launched.

The following software versions are available:

- Control Expert S
- Control Expert L
- Control Expert XL
- Control Expert XL with M580 Safety

The M580 Safety CPUs are included in a **Safety add-on** available for Control Expert L and XL.

Two types of licenses are available to activate Control Expert:

- Node-locked license for single use on a local PC.
- Floating license for multiple uses of an authorized number of PCs in a network connected to the Enterprise License Server.

For detailed information on license activation and/or registration, refer to *EcoStruxure™ Control Expert, Installation Manual*.

Performance Scope

This table shows the main characteristics of the individual software versions:

| | Control Expert S | Control Expert L | Control Expert L + Safety add-on | Control Expert XL | Control Expert XL + Safety add-on |
|--------------------------------|------------------|------------------|----------------------------------|-------------------|-----------------------------------|
| Programming languages | | | | | |
| Function Block Diagram FBD | + | + | + | + | + |
| Ladder Diagram (LD) language | + | + | + | + | + |
| Instruction List IL | + | + | +(2) | + | +(2) |
| Structured Text ST | + | + | +(2) | + | +(2) |
| Sequential Language SFC | + | + | +(2) | + | +(2) |
| Ladder Logic 984 (LL984) | + | + | + | + | + |
| Libraries⁽¹⁾ | | | | | |
| Standard library | + | + | +(2) | + | +(2) |

| | Control Expert S | Control Expert L | Control Expert L + Safety add-on | Control Expert XL | Control Expert XL + Safety add-on |
|--|------------------------------------|------------------------------------|------------------------------------|------------------------------------|------------------------------------|
| Control library | + | + | +(2) | + | +(2) |
| Communication library | + | + | +(2) | + | +(2) |
| Diagnostics library | + | + | +(2) | + | +(2) |
| I/O management library | + | + | +(2) | + | +(2) |
| System library | + | + | +(2) | + | +(2) |
| Motion control drive library | - | + | +(2) | + | +(2) |
| TCP Open library | - | optional | optional ⁽²⁾ | optional | optional ⁽²⁾ |
| Obsolete library | + | + | +(2) | + | +(2) |
| MFB library | + | + | +(2) | + | +(2) |
| Safety library | - | - | + | - | + |
| Memory card file management library | + | + | +(2) | + | +(2) |
| General information | | | | | |
| Create and use data structures (DDTs) | + | + | +(2) | + | +(2) |
| Create and use Derived Function Blocks (DFBs) | + | + | + | + | + |
| Project browser with structural and/or functional view | + | + | + | + | + |
| Managing access rights | + | + | + | + | + |
| Operator screen | + | + | + | + | + |
| Diagnostic viewer | + | + | + | + | + |
| System diagnostics | + | + | + | + | + |
| Project diagnostics | + | + | +(2) | + | +(2) |
| Trending Tool | + | + | + | + | + |
| Application converter | PL7 converter Concept Converter | PL7 converter Concept Converter | PL7 converter Concept Converter | PL7 converter Concept Converter | PL7 converter Concept Converter |

| | Control Expert S | Control Expert L | Control Expert L + Safety add-on | Control Expert XL | Control Expert XL + Safety add-on |
|----------------------------|---|---|--|--|---|
| | Partial conversion | | | | |
| Managing multi-stations | - | - | - | - | - |
| Supported platforms | | | | | |
| Modicon M340 | All CPUs | All CPUs | All CPUs | All CPUs | All CPUs |
| Modicon M580 | - | BMEP5810** BMEP5820** BMEP5830** BMEH582040 | BMEP5810** BMEP5820** BMEP5830** BMEH582040 BMEP582040S BMEH582040S | BMEP5810** BMEP5820** BMEP5830** BMEP5840** BMEP585040 BMEP586040 BMEH582040 BMEH584040 BMEH586040 | BMEP5810** BMEP5820** BMEP5830** BMEP5840** BMEP585040 BMEP586040 BMEH582040 BMEH584040 BMEH586040 BMEP582040S BMEP584040S BMEH582040S BMEH584040S BMEH586040S |
| Momentum | 171CBU78090 171CBU98090 171CBU98091 | 171CBU78090 171CBU98090 171CBU98091 | 171CBU78090 171CBU98090 171CBU98091 | 171CBU78090 171CBU98090 171CBU98091 | 171CBU78090 171CBU98090 171CBU98091 |
| Premium | - | All CPUs except: P57 554M P57 5634M P57 6634M | All CPUs except: P57 554M P57 5634M P57 6634M | All CPUs | All CPUs |
| Quantum | - | 140CPU31110 140CPU43412 U/A* | 140CPU31110 140CPU43412 U/A* | 140CPU31110 140CPU43412 U/A | 140CPU31110 140CPU43412 U/A |

| | Control Expert S | Control Expert L | Control Expert L + Safety add-on | Control Expert XL | Control Expert XL + Safety add-on |
|---|------------------|---|---|--|--|
| | | 140CPU53414 U/A* * Upgrade using OS Loader | 140CPU53414 U/A* * Upgrade using OS Loader | 140CPU53414 U/A 140CPU65150 140CPU65160 140CPU65260 140CPU65860 140CPU67060 140CPU67160 140CPU67260 140CPU67261 140CPU67861 | 140CPU53414 U/A 140CPU65150 140CPU65160 140CPU65260 140CPU65860 140CPU67060 140CPU67160 140CPU67260 140CPU67261 140CPU67861 |
| Atrium | - | All CPUs | All CPUs | All CPUs | All CPUs |
| Simulator | + | + | + | + | + |
| Openness | | | | | |
| Hyperlinks | + | + | + | + | + |
| Control Expert Server (for OFS, UAG) | + | + | + | + | + |
| Software components contained in the software package | | | | | |
| Documentation as context help and PDF | + | + | + | + | + |
| OS Loader tool + HW firmware | + | + | + | + | + |
| + Available - Not available (1) Availability of the blocks depends on the hardware platforms. (2) Available on all PLC except platforms M580 Safety. | | | | | |

Naming Convention

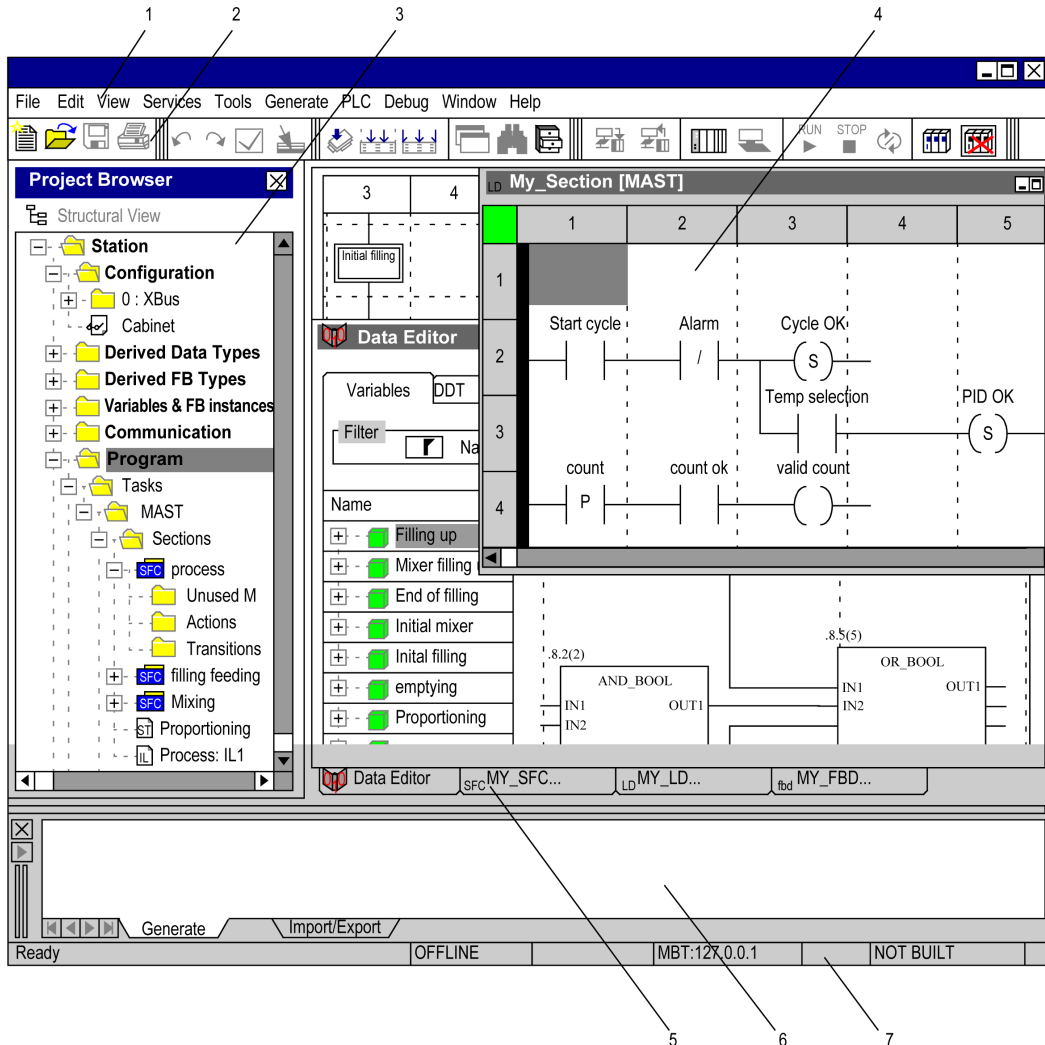
In the rest of this document, "Control Expert" is used as general term for "Control Expert S", "Control Expert L", and "Control Expert XL", with or without Safety add-on.

User Interface

Overview

The user interface consists of several, configurable windows and toolbars.

User interface:



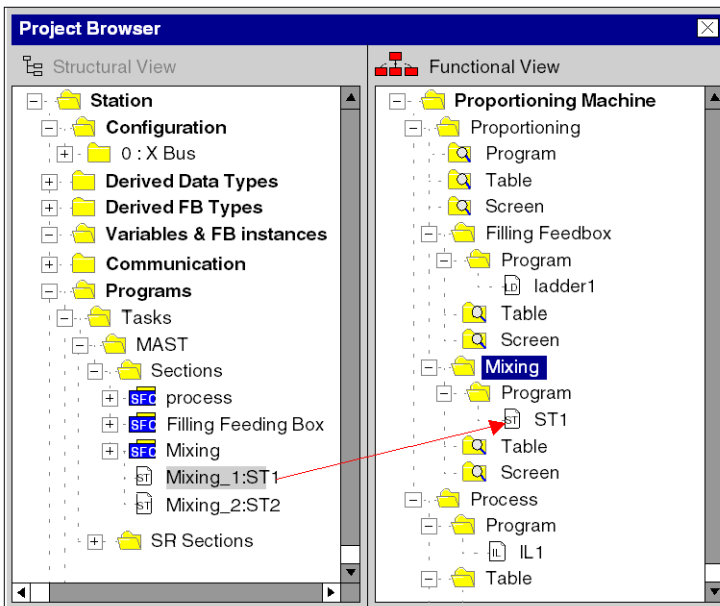
Legend:

| Number | Description |
|--------|--|
| 1 | Menu bar (see EcoStruxure™ Control Expert, Operating Modes) |
| 2 | Toolbar (see EcoStruxure™ Control Expert, Operating Modes) |
| 3 | Project Browser (see EcoStruxure™ Control Expert, Operating Modes) |
| 4 | Editor window (programming language editors, data editor, etc.) |
| 5 | Register tabs for direct access to the editor window |
| 6 | Information window (see EcoStruxure™ Control Expert, Operating Modes) (provides information about errors which have occurred, signal tracking, import functions, etc.) |
| 7 | Status bar (see EcoStruxure™ Control Expert, Operating Modes) |

Project Browser

Introduction

The **Project Browser** displays all project parameters. The view can be shown as structural (topological) and/or functional view.



Structural View

The project browser offers the following features in the structural view:

- Creation and deletion of elements
- Symbol showing if sections and Program Units are protected.
- The section symbol shows the section programming language (in case of an empty section the symbol is grey)
- View the element properties
- Creation of user directories
- Launching the different editors
- Start the import/export function

Functional View

The project browser offers the following features in functional view:

- Creation of functional modules
- Insertion of Program Units, sections, animation tables etc. using drag and drop from the structural view
- Creation of program elements (Program Units, sections)
- View the element properties
- Launching the different editors
- The section symbol shows the section programming language and other attributes

User Application and Project File Formats

Introduction

Control Expert manages four types of files for storing user applications and projects. Each type of file can be used according to specific requirements.

File types can be identified by their extension:

- *.STU: File
- *.STA: Archived Application File
- *.XEF: Application Exchange File
- *.ZEF: Full Application Exchange File

STU File

This file type is used for daily working tasks. This format is used by default when opening or saving a user project.

The following table presents the *STU* file advantages and drawbacks:

| Advantages | Drawbacks |
|---|--|
| <ul style="list-style-type: none"> The project can be saved at any stage (consistent or inconsistent) through the default command. | <ul style="list-style-type: none"> Not convenient when transferring project due to the very large size of the file. |
| <ul style="list-style-type: none"> Project saving and opening is fast as the entire internal database is present in the file. | <ul style="list-style-type: none"> Not compatible when updating Control Expert from one version to another. |
| <ul style="list-style-type: none"> Automatic creation of BAK files¹ | |

¹ Each time a **STU** file is saved, a backup copy is also created, with the same name as the **STU** file, and the extension **BAK** files. By changing the file extension from **BAK** to **STU**, it is possible to revert to the state the project was, the last time it was saved. **BAK** files are stored in the same folder as the project **STU** file.

STA File

This file type is used for archiving projects and can be created only after the project has been generated. This file type allows forward compatibility between the different versions of Control Expert.

There are 2 ways to create an **STA** file:

- STA** file can be created **manually** by accessing the **File > Save Archive** menu in the Control Expert main window.
- STA** file is created **automatically** every time the project is saved as a **STU** file if it is in **Built** state.

NOTE: The STA file created automatically is saved into the same directory and with the same file name as the STU project file, except that a **“.Auto”** suffix is appended to the file name. If an existing automatic STA file already exists, **it is overwritten** without any confirmation.

NOTE: If the project is in **Built** state, saving a STU file through a Control Expert Server creates a STA file as well.

The following table presents the *STA* file advantages and drawbacks:

| Advantages | Drawbacks |
|---|--|
| <ul style="list-style-type: none"> Fast project saving. | <ul style="list-style-type: none"> Can be created only after the project has been generated. |
| <ul style="list-style-type: none"> Projects can be shared via e-mail or low size memory supports. | <ul style="list-style-type: none"> Opening of the project is long, as the project file is rebuilt before operation. |
| <ul style="list-style-type: none"> Capability to connect in Equal Online Mode to the PLC after opening the project on a new version of Control Expert. For additional information, see Connection/Disconnection (see EcoStruxure™ Control Expert, Operating Modes). | |
| <ul style="list-style-type: none"> Allow online modifications with the PLC without any prior download into the PLC. | |
| <ul style="list-style-type: none"> Generated <i>STA</i> file is compatible with all Control Expert versions. NOTE: In order to load a <i>STA</i> file created with another version of Control Expert, all the features used in the application have to be supported by the current version. | |

Create Archived Application File (*.STA)

The following table describes the procedure for generating *.STA files:

| Step | Action |
|------|---|
| 1 | Launch the current Control Expert software: Start > Programs > EcoStruxure Control Expert > Control Expert. |
| 2 | Open the project (*.STU file): <ol style="list-style-type: none"> File > Open. Select the project (*.STU file). Click Open. |
| 3 | File > Save Archive , see note below. |
| 4 | Choose a location for the file to be saved. Do not save files in the default Schneider Electric directory: <i>C:\Program Files\Schneider Electric\Control Expert</i> Files saved in this directory may be deleted during Control Expert installation. |

| Step | Action |
|------|---|
| 5 | Click Save . |
| 6 | Remember the location where the *.STA file is stored on the terminal as it is needed when recovering the project. |

NOTE: The **Save Archive** function is only available if:

- The project has been generated.
- In **Tools > Project Settings**, in the **Upload Information** section if **Include** is selected, at least one of the two check boxes underneath must be checked.

Restoring Archived Application File (*.STA)

This restoration consists of importing the *.STA files previously created and stored. The *.STA files are used when the PLC cannot be stopped. To restore *.STA files follow the procedure below for each project:

| Step | Action |
|------|---|
| 1 | Launch Control Expert: Start > Programs > EcoStruxure Control Expert > Control Expert. |
| 2 | Open the *.STA file from File > Open menu. The file type selected must be <i>Archived Application File (STA)</i> . |
| 3 | Click Open . |
| 4 | Save the project as an *.STU file. |

XEF File

This file type is used for exporting projects in an *XML* source format and can be created at any stage of a project.

The following table presents the **XEF** file advantages and drawbacks:

| Advantages | Drawbacks |
|--|---|
| <ul style="list-style-type: none"> • The <i>XML</i> source format ensures project compatibility with any version of Control Expert. | <ul style="list-style-type: none"> • Medium size. |
| | <ul style="list-style-type: none"> • Opening of the project takes time while the project is imported before operation. |

| Advantages | Drawbacks |
|------------|--|
| | <ul style="list-style-type: none"> • Generation of the project is mandatory to re-assemble the project binary code. |
| | <ul style="list-style-type: none"> • Operating with the PLC requires to rebuild all the project and perform a download in the processor. |
| | <ul style="list-style-type: none"> • Connecting to the PLC in Equal Online mode with an XEF file is not possible. For additional information, see Connection/Disconnection (see EcoStruxure™ Control Expert, Operating Modes). |

ZEF File

This file type is used for exporting projects with global DTMs configuration and can be created at any stage of a project. For details on project export/import, refer to chapter **Import / Export** (see EcoStruxure™ Control Expert, Operating Modes).

The following table presents the **ZEF** file advantages and drawbacks:

| Advantages | Drawbacks |
|---|---|
| <ul style="list-style-type: none"> • The ZEF format ensures project compatibility with any version of Control Expert. | <ul style="list-style-type: none"> • Medium size. |
| | <ul style="list-style-type: none"> • Opening of the project takes time while the project is imported before operation. |
| | <ul style="list-style-type: none"> • Generation of the project is mandatory to re-assemble the project binary code. |
| | <ul style="list-style-type: none"> • Operating with the PLC requires to rebuild all the project and perform a download in the processor. |
| | <ul style="list-style-type: none"> • Connecting to the PLC in Equal Online mode with a ZEF file is not possible. For additional information, see Connection/Disconnection (see EcoStruxure™ Control Expert, Operating Modes). |

Create Application Exchange File (*.ZEF or *.XEF)

The following table describes the procedure for generating *.ZEF or *.XEF files:

| Step | Action |
|------|---|
| 1 | Launch the current Control Expert software: Start > Programs > EcoStruxure Control Expert > Control Expert. |
| 2 | Open the project (*.STU file): 1. File > Open. 2. Select the project (*.STU file). 3. Click Open. |
| 3 | File > Export Project. |
| 4 | Choose a location for the file to be saved. Do not save files in the default Schneider Electric directory: <i>C:\Program Files\Schneider Electric\Control Expert</i> Files saved in this directory may be deleted during Control Expert installation. |
| 5 | Click Export and select the export file format (*.ZEF or *.XEF). |
| 6 | Remember the location where the *.ZEF or *.XEF file is stored on the workstation as it is needed when recovering the project. |

Restoring Application Exchange File (*.ZEF or *.XEF)

This restoration consists of importing the *.ZEF or *.XEF files previously created and stored. Importing from a ZEF or XEF format involves the re-generation of the project. To restore *.ZEF or *.XEF files follow the procedure below for each project:

| Step | Action |
|------|---|
| 1 | Launch Control Expert: Start > Programs > EcoStruxure Control Expert > Control Expert. |
| 2 | Open the *.ZEF or *.XEF file from File > Open menu. The file type selected must be <i>Full Application Exchange File (*.ZEF)</i> or <i>Application Exchange File (*.XEF)</i> . |
| 3 | Click Open. |
| 4 | Save the project as an *.STU file. |

Compatibility Information

The **STU** files are not compatible across Control Expert versions. In order to use a project with different Control Expert versions, users must either store, the:

- Archived Application Files (STA):

With the **STA** file, it is possible to reuse the current built project with the new Control Expert version installed on the computer.

- Application Exchange Files (*ZEF*):
The **ZEF** file must be used if the project has been built.
- Application Exchange Files (*XEF*):
The **XEF** file must be used if the project has been built.

Comparative File Types

The following table gives a summary of the 4 file types:

| File Types | <i>STU</i> | <i>STA</i> | <i>XEF</i> | <i>ZEF</i> |
|--|-------------|-------------------|-----------------------|-----------------------|
| Binary applications | Yes | Yes | No | No |
| Source applications | Yes | Yes | Yes | Yes |
| Internal database | Yes | No | No | No |
| Comparative file size | 10, see (1) | 0.03, see (1) | 3 | 3 |
| Comparative time to save | 10 | 1.6 | 6 | 6 |
| Comparative time to open | 1 | 10 | 10 | 10 |
| Connection to the PLC in Equal Online mode | Possible | Possible | Not possible, see (2) | Not possible, see (2) |
| File backup | Possible | Possible, see (3) | Possible | Possible |

(1): Compressed files.

(2): The project needs to be first uploaded into the PLC.

(3): The project can be saved only if it has been generated.

NOTE: The values in the table represent a ratio between file types, where the *STU* value is the reference.

Configurator

Configurator Window

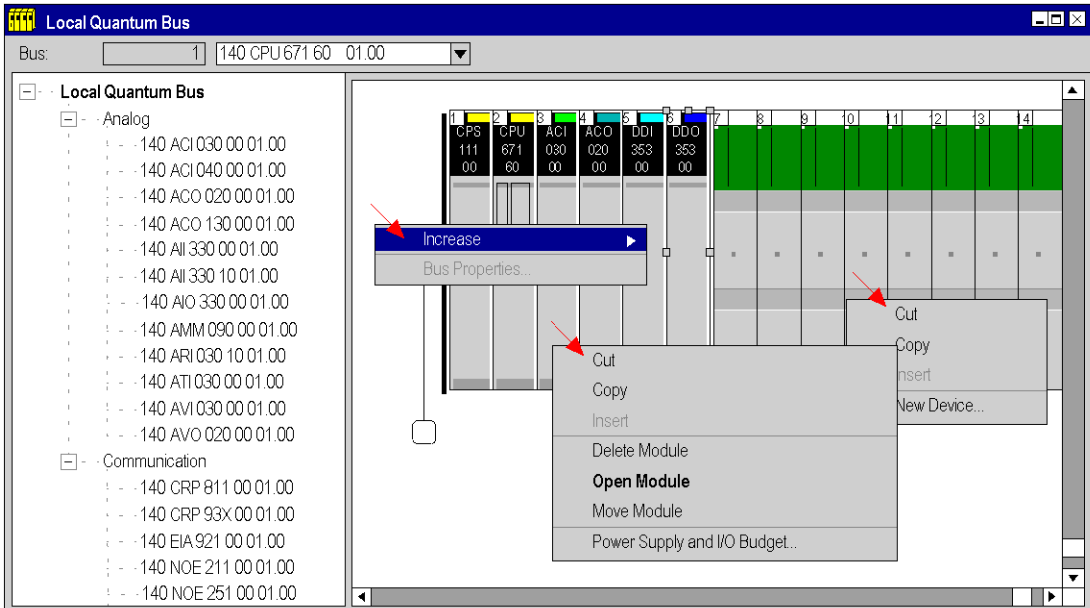
The configurator window is split into two windows:

- Catalog window

A module can be selected from this window and directly inserted in the graphical representation of the PLC configuration by dragging and dropping.

- Graphical representation of the PLC configuration

Representation of the Configurator window:



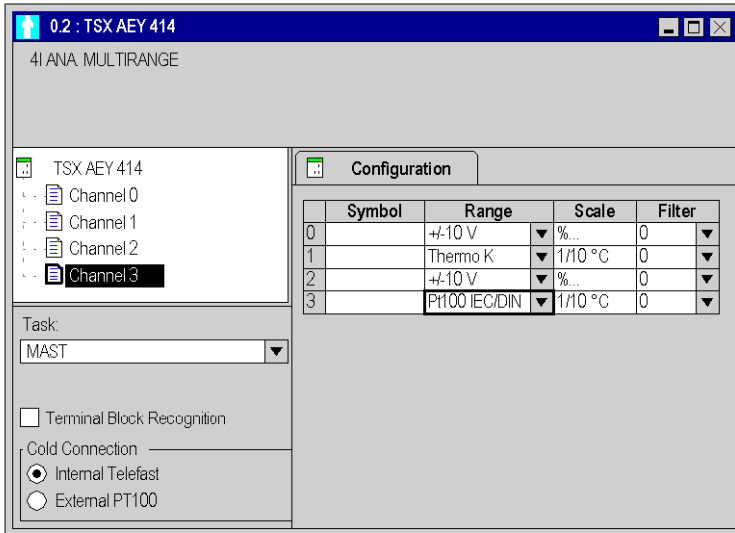
One of the following shortcut menus is called depending on the position of the mouse pointer:

- Mouse pointer on the background allows among others:
 - Change CPU,
 - Selection of different Zoom factors.
- Mouse pointer on the module allows among others:
 - Access to editor functions (delete, copy, move),
 - Open the module configuration for defining the module specific parameters,
 - Show the I/O properties and the total current.
- Mouse pointer on an empty slot allows among others:
 - Insert a module from the catalog,
 - Insert a previously copied module including its defined properties.

Module Configuration

The module configuration window (called via the modules shortcut menu or a double-click on the module) is used to configure the module. This also includes channel selection, selection of functions for the channel selected, assignment of State RAM addresses (only Quantum) etc.

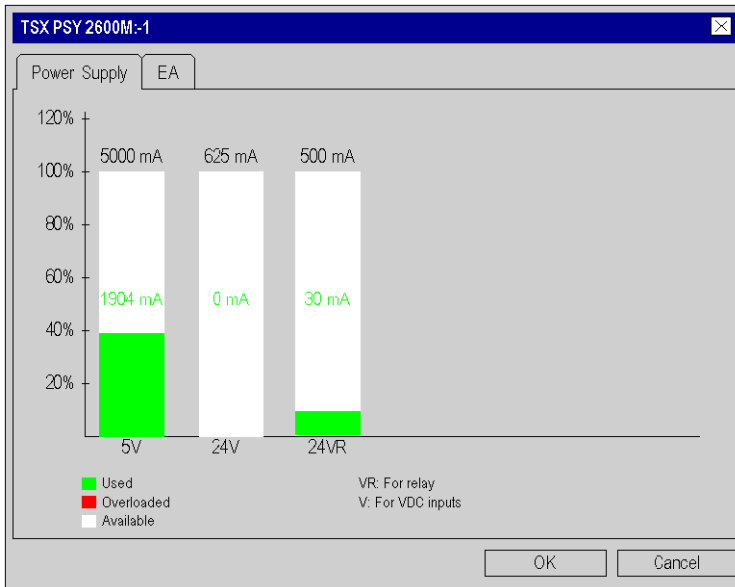
Module configuration window for a Premium I/O module:



Module Properties

The module properties window (called via the modules shortcut menu) shows the modules properties such as the power consumption, number of I/O points (only Premium) and more.

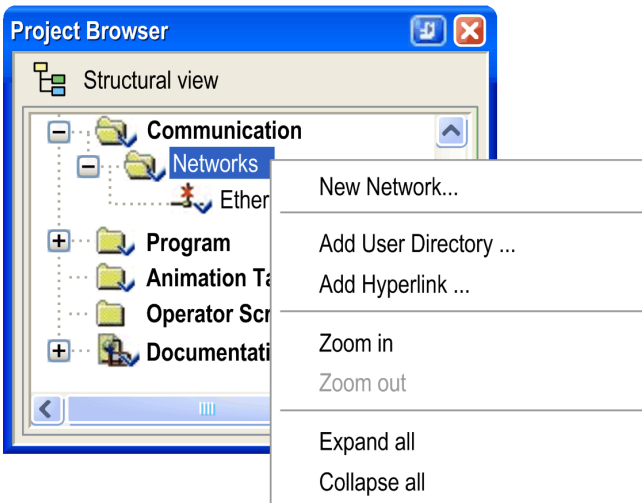
The module properties window for the power supply shows the total current of the rack:



Network Configuration

The network configuration is called via the communications folder.

Network configuration:



The network configuration windows allow among others:

- Creation of networks
- Network analysis
- Printout of the network configuration

A window for configuring a network:

The screenshot shows the 'Ethernet_1' configuration window. The 'Model Family' is set to 'TCP/IP 10/100 Regular connection'. The 'Module Address' section has empty fields for Rack, Module, and Channel. The 'Module Utilities' section has dropdown menus for IO Scanning (NO), Global Data (NO), SNMP (YES), Address Server (NO), and NTP (NO). The 'Module IP Address' section has fields for IP Address (0.0.0.0), Subnetwork Mask (255.0.0.0), and Gateway Address (0.0.0.0). The 'IP Configuration' tab is active, showing the 'Configured' radio button selected and fields for IP address (0.0.0.0), Subnetwork mask (255.0.0.0), and Gateway address (0.0.0.0). The 'Ethernet configuration' section has the 'Ethernet II' radio button selected.

After configuration the network is assigned a communications module.

Data Editor

Introduction

The data editor offers the following features:

- Declaration of variable instances
- Definition of derived data types (DDTs)
- Definition of Device derived data types (Device DDTs)
- Instance declaration of elements and derived function blocks (EFBs/DFBs)
- Definition of derived function block (DFBs) parameters

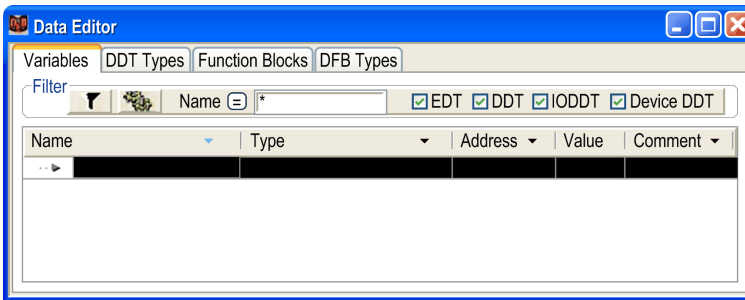
The following functions are available in all tabs of the data editor:

- **Copy, Cut, Paste** with the following restrictions:
 - **Edit > Cut** menu command is greyed in every tabs.
 - Right-click **Cut** on a variable is greyed in every tabs
 - **Edit > Copy** and **Edit > Paste** menu commands are not greyed but are not working in **DDT Types** and **DFB Types** tabs.
 - Right-click **Copy** and right-click **Paste** on a variable are greyed in **DDT Types**, **Function Blocks**, and **DFB Types** tabs.
- Expand/collapse structured data
- Sorting according to Type, Symbol, Address etc.
- Filter
- Inserting, deleting and changing the position of columns
- Drag and Drop between the data editor and the program editors
- Undo the last change
- Export/Import

Variables

The **Variables** tab is used for declaring variables.

Variables tab:



The following functions are available:

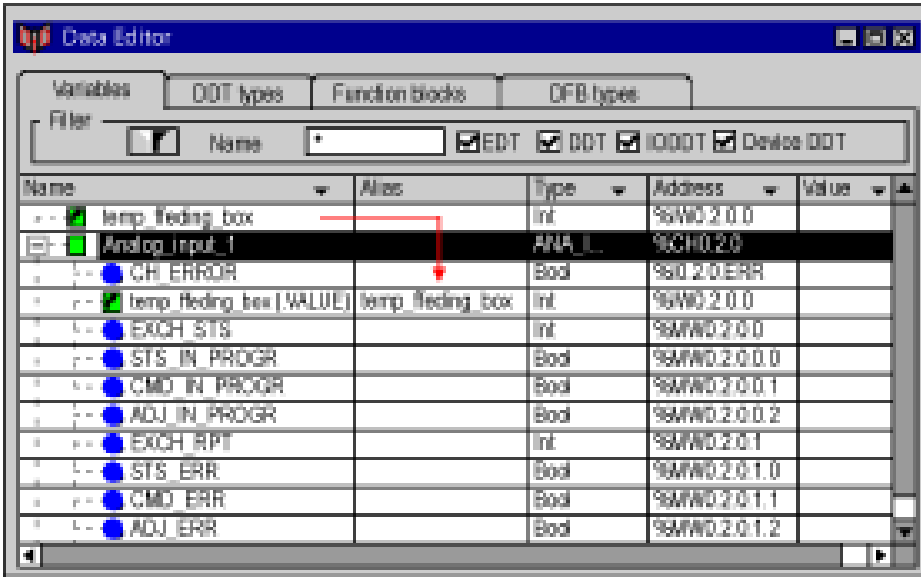
- Defining a symbol for variables
- Assigning data types
- Own selection dialog box for derived data types
- Assignment of an address
- Automatic symbolization of I/O variables

- Assignment of an initial value
- Entering a comment
- View all properties of a variable in a separate properties dialog box

Hardware Dependent Data Types (IODDT)

IODDTs are used to assign the complete I/O structure of a module to an individual variable.

Assignment of IODDTs:



The following functions are available:

- Complete I/O structures can be assigned with individual variables using IO DDTs
- After entering the variables addresses, all elements of the structure are automatically assigned with the correct input/output bit or word
- Because it is possible to assign addresses later on, standard modules can be simply created whose names are defined at a later date.
- An alias name can be given to all elements of an IODDT structure.

Hardware Dependent Device Derived Data Types (Device DDT)

Device derived data type (DDT) is a predefined DDT that describes the I/O language elements of an I/O Module. This data type is represented in a structure, which depends on the capabilities of the I/O module.

This structure provides bits and register views when both extracted bits and register exist in IODDT. In this case extracted bit is not seen as a child element of the register variable but directly as field of Device DDT structure.

When adding a Modicon M340 module in a M340 remote I/O drop the Control Expert software will create automatically the associated Device DDT instance. This instance is deduced from IODDT (other not mapped object like %KW are not accessible).

Each I/O Module is associated with one implicit device DDT instance:

- Implicit Device DDT instances are created by default on device insertion and refreshed automatically by the PLC. They contain the modules status, modules and channels health bits, values of the modules inputs, values of the modules outputs, etc.

The Implicit Device DDT can be:

- linked to a device (Managed)
- not linked to a device (Un-managed)

NOTE: IODDT and topologic address (see Modicon M340, CANopen, User Manual) are no longer supported with the Modicon M340 remote I/O modules. All the informations (bits and registers) related to a channel are accessible directly as a field of device DDT structure.

NOTE: Optional Explicit structures are DDT Explicit DDT, created on demand from data editor and used through Function block to be refreshed.

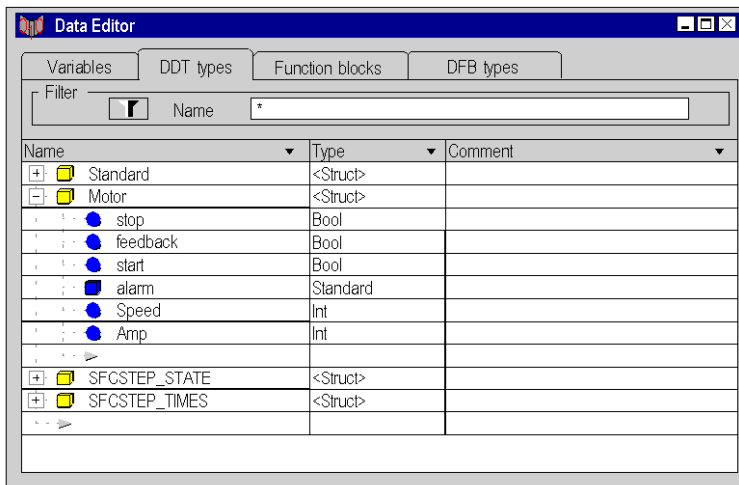
NOTE: Optional Explicit DDT types are proposed in the data editor to support Status or Command data used by explicit exchanges on a Channel of an IO Modules in a M340 remote I/O drop. Explicit DDT instances are created manually by the user in the data editor and used as Input or Output Parameter with the Function block managing the explicit exchange (READ_STS_QX (see EcoStruxure™ Control Expert, I/O Management, Block Library), WRITE_CMD_QX (see EcoStruxure™ Control Expert, I/O Management, Block Library)).

Derived Data Types (DDT)

The **DDT types** tab is used for defining derived data types (DDTs).

A derived data type is the definition of a structure or array from any data type already defined (elementary or derived).

Tab DDT types:



The following functions are available:

- Definition of nested DDTs (max. 15 levels)
- Definition of arrays with up to 6 dimensions
- Assignment of an initial value
- Assignment of an address
- Entering a comment
- Analysis of derived data types
- Assignment of derived data types to a library
- View all properties of a derived data type in a separate properties dialog box
- An alias name can be given to all elements of a DDT structure or an array.

Function Blocks

The **Function blocks** tab is used for the instance declaration of elements and derived function blocks (EFBs/DFBs).

Tab Function blocks:

| Name | Number | Type | Value | Comment |
|------------|--------|------------|-------|-------------------|
| SFCControl | | SFCCNTRL | | |
| <Inputs> | | | | |
| CHARTREF | 1 | SFCG-AR... | | Link to SFC |
| INIT | 2 | Bool | FALSE | Reset SFC |
| CLEAR | 3 | Bool | FALSE | Reset SFC |
| DISTIME | 4 | Bool | FALSE | Supervision time |
| DISTRANS | 5 | Bool | FALSE | Transitions |
| DISACT | 6 | Bool | FALSE | Action processing |
| STEPUN | 7 | Bool | FALSE | Step over, |
| STEPDEP | 8 | Bool | FALSE | StepOver |
| RESTERR | 9 | Bool | FALSE | Supervision time |
| DISRMOTE | 10 | Bool | FALSE | Remote control |
| ALLTRANS | 11 | Bool | FALSE | All transitions |
| RESSTEPT | 12 | Bool | FALSE | Elapsed time |

The following functions are available:

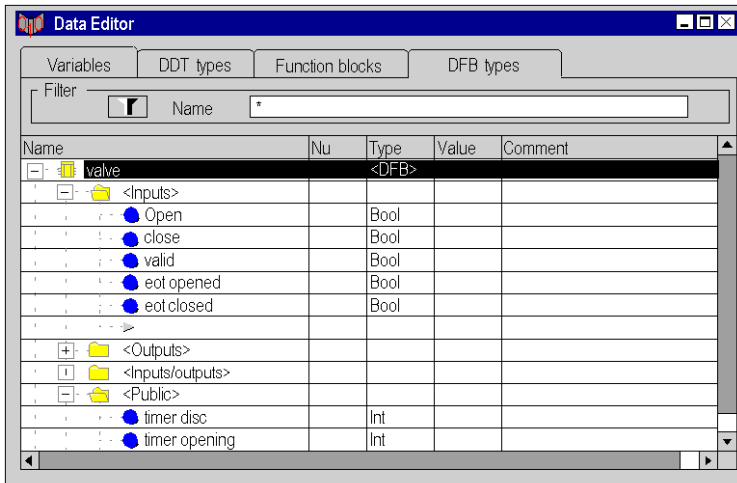
- Display of the function blocks used in the project
- Definition of a symbol for the function blocks used in the project
- Automatic enabling of the defined symbols in the project
- Enter a comment about the function block
- View all parameters (inputs/outputs) of the function block
- Assignment of an initial value to the function block inputs/outputs

DFB Types

The **DFB types** tab is used for the defining derived function block (DFBs) parameters.

The creation of DFB logic is carried out directly in one or more sections of the FBD, LD, IL or ST programming languages.

Tab DFB types:



The following functions are available:

- Definition of the DFB name
- Definition of all parameter of the DFB, such as:
 - Inputs
 - Outputs
 - VAR_IN_OUT (combined inputs/outputs)
 - Private variables
 - Public variables
- Assignment of data types to DFB parameters
- Own selection dialog box for derived data types
- Assignment of an initial value
- Nesting DFBs
- Use of several sections in a DFB
- Enter a comment for DFBs and DFB parameters
- Analyze the defined DFBs
- Version management
- Assignment of defined DFBs to a library

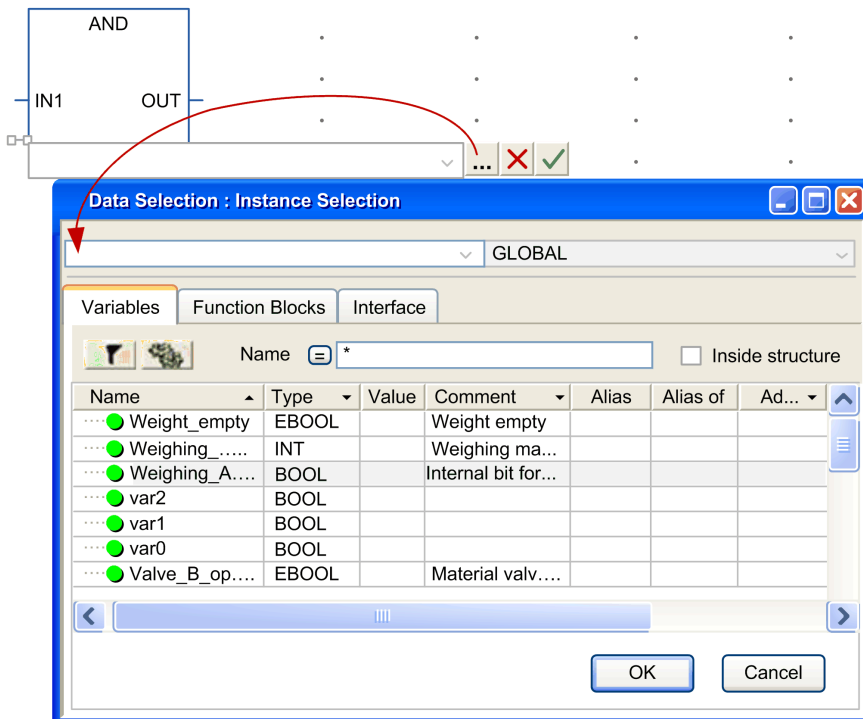
Data Usage

Data types and instances created using the data editor can be inserted (context dependent) in the programming editors.

The following functions are available:

- Access to all programming language editors
- Only compatible data is displayed
- View of the data according to their scope affiliation
- View of the functions, function blocks, procedures and derived data types arranged according to their library affiliation
- Instance declaration during programming is possible

Data selection dialog box:



Online Modifications

It is possible to modify the type of a variable or a Function Block (FB) instance declared in application or in a Derived Function Block (DFB) directly in online mode (see EcoStruxure™

Control Expert, Operating Modes). That means it is not required to stop the application to perform such a type modification.

These operations can be done either in the data editor or in the properties editor, in the same way as in offline mode.

When changing the type of a variable, the new value of the variable to be modified depends on its kind:

- In the case of an **unlocated variable**, the variable is set to the initial value, if one exists. Otherwise, it is set to the default value.
- In the case of a **located variable**, the variable restarts with the initial value if one exists. Otherwise, the current binary value is unchanged.

CAUTION

UNEXPECTED APPLICATION BEHAVIOR

Before applying the variable type change, check the impact of the new value of the variable on the application execution.

Failure to follow these instructions can result in injury or equipment damage.

NOTE: It is not possible to modify the type of a variable declared in Derived Data Type (DDT) in online mode (see EcoStruxure™ Control Expert, Operating Modes). The application has to be switched into offline mode (see EcoStruxure™ Control Expert, Operating Modes) in order to build such a modification.

Restrictions About Online Modifications

In the following cases, the online type modification of a variable or of a Function Block (FB) is not allowed:

- If the variable is used as network global data, the online type modification is not permitted.
- Whether the current FB can not be removed online, or a new FB can not be added online, the online type modification of this FB is not allowed. Indeed, some Elementary Function Blocks (EFB) like the Standard Function Blocks (SFB) do not allow to be added or removed online. As a result, changing the type of an EFB instance to a SFB instance is not possible, and conversely.

In both of these cases, the following dialog box is displayed:



NOTE: Due to these limitations, if a Derived Function Block (DFB) contains at least one instance of a SFB, it is not possible to add or remove instance of this DFB in online mode (see EcoStruxure™ Control Expert, Operating Modes).

Program Unit Data Editor

Introduction

NOTE: The Program Unit data editor is a data editor limited to the scope of the Program Unit to which it belongs.

The Program Unit data editor offers the following features:

- Declaration of variables associated to the Program Unit
- Instance declaration of elementary and derived function blocks (EFBs/DFBs) used in the Program Unit
- Management of the Program Unit parameters

The following functions are available in all tabs of the data editor:

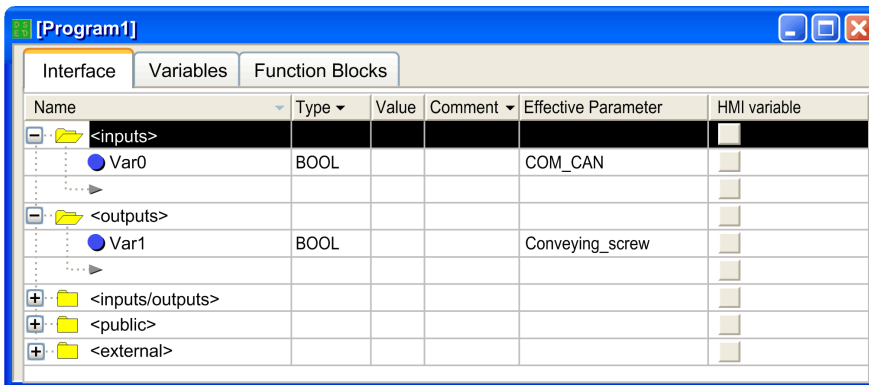
- **Copy, Cut, Paste** with the following restrictions:
 - **Edit > Cut** menu command is greyed in every tabs.
 - Right-click **Cut** on a variable is greyed in every tabs
 - **Edit > Copy** and **Edit > Paste** menu commands are not greyed but are not working in **Interface** tab.
 - Right-click **Copy** and right-click **Paste** on a variable are greyed in **Interface**, and **Function Blocks** tabs.
- Expand/collapse structured data
- Sorting according to Symbol, Type, etc.
- Filter
- Inserting, deleting and changing the position of columns
- Drag and drop between the Program Unit data editor and the program editors

- Undo the last change
- Export/Import

Variables associated to a Program Unit can be:

- Private: can only be R/W in the scope of this Program Unit.
- Public: can be R/W out of the scope of this Program Unit.
- Parameters (inputs, outputs, inputs/outputs): linked to Public variables (from an other Program Unit), or Global variables.

Interface Tab



The **Interface** tab is used to manage:

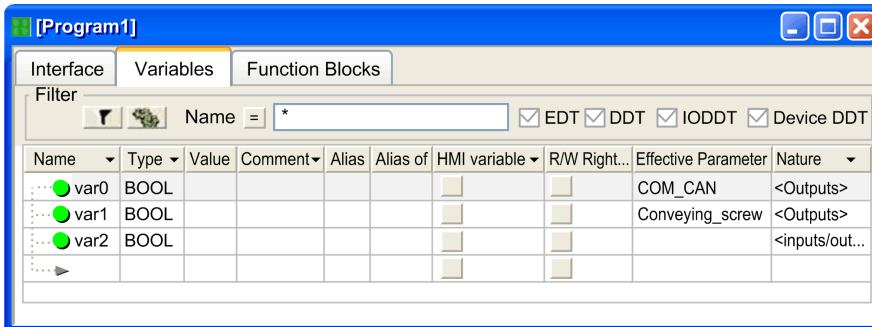
- input, output and input/output parameters.
- public variables
- external variables

The following functions are available:

- Assigning of an effective parameter (can be a Global variable or a Public variable from another Program Unit) to input, output and input/output parameters.
- Assigning of an initial value
- Defining a symbol for parameters and variables
- Assigning data types
- Displaying all properties of a variable in a separate properties dialog box
- Entering a comment

Variables

The **Variables** tab is used for declaring variables used by the Program Unit:



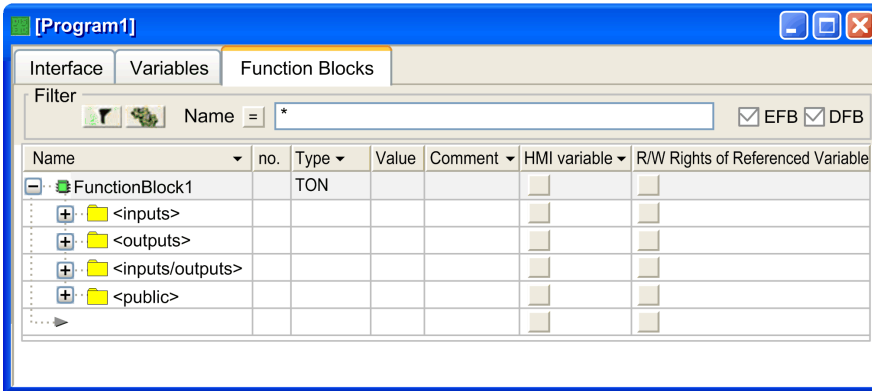
The following functions are available:

- Defining a symbol for variables
- Assigning data types
- Assigning variables to Program Unit parameters (**Effective Parameter**)
- Declaring the **Nature** of the variables.
- Own selection dialog box for data types
- Assignment of an initial value
- Entering a comment
- Displaying all properties of a variable in a separate properties dialog box

Function Blocks

The **Function blocks** tab is used for the instance declaration of elements and derived function blocks (EFBs/DFBs).

Tab **Function blocks**:



The following functions are available:

- Displaying the function blocks used in the Program Unit
- Defining symbol for the function blocks used in the Program Unit
- Automatic enabling of the defined symbols in the Program Unit
- Entering a comment for the function block
- Displaying all parameters (inputs/outputs) of the function block
- Assigning of initial values to the function block inputs/outputs

Program Unit Data Usage

Variables and instances created using the program data editor can be inserted (context dependent) in the programming editors using the **Instance Selection** dialog box.

NOTE: Variables and instances can also be created on the fly in the different language editors.

Online Modifications

It is possible to modify the type of a variable or a Function Block (FB) instance declared in application or in a Derived Function Block (DFB) directly in online mode (see EcoStruxure™ Control Expert, Operating Modes). That means it is not required to stop the application to perform such a type modification.

These operations can be done either in the data editor or in the properties editor, in the same way as in offline mode.

When changing the type of a variable, the new value of the variable to be modified depends on its kind:

- In the case of an **unlocated variable**, the variable is set to the initial value, if one exists. Otherwise, it is set to the default value.
- In the case of a **located variable**, the variable restarts with the initial value if one exists. Otherwise, the current binary value is unchanged.

⚠ CAUTION

UNEXPECTED APPLICATION BEHAVIOR

Before applying the variable type change, check the impact of the new value of the variable on the application execution.

Failure to follow these instructions can result in injury or equipment damage.

NOTE: When a reference (REF_TO) is used as **Effective Parameter**, if the effective parameter is deleted then the REF_TO value in Program Unit is updated to its initial value after controller INIT.

Restrictions About Online Modifications

In the following cases, the online type modification of a variable or of a Function Block (FB) is not allowed:

- If the variable is used as network global data, the online type modification is not permitted.
- Whether the current FB can not be removed online, or a new FB can not be added online, the online type modification of this FB is not allowed. Indeed, some Elementary Function Blocks (EFB) like the Standard Function Blocks (SFB) do not allow to be added or removed online. As a result, changing the type of an EFB instance to a SFB instance is not possible, and conversely.

In both of these cases, the following dialog box is displayed:



NOTE: Due to these limitations, if a Derived Function Block (DFB) contains at least one instance of a SFB, it is not possible to add or remove instance of this DFB in online mode (see EcoStruxure™ Control Expert, Operating Modes).

Program Editor

Introduction

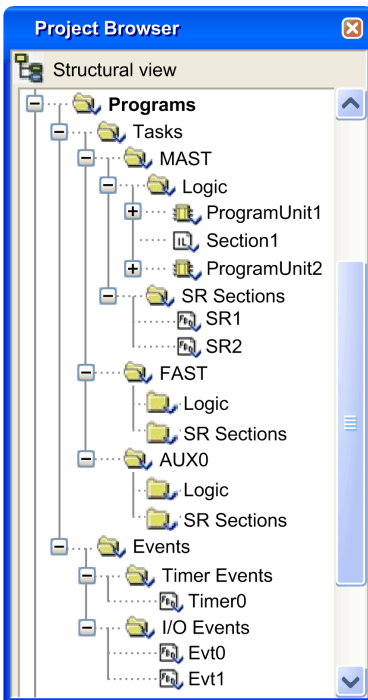
A program can be built from:

- **Tasks**, that are executed cyclically or periodically.
Tasks are built from:
 - Program Units (only for Modicon M580 and M340)
 - Sections
 - Subroutines
- **Event processing**, that is carried out before all other tasks.

Event processing is built from:

- Sections for processing time controlled events
- Sections for processing hardware controlled events

Example of a program:



Tasks

Control Expert supports multiple tasks (Multitasking).

The tasks are executed in parallel and independently of each other whereby the execution priorities are controlled by the PLC. The tasks can be adjusted to meet various requirements and are therefore a powerful instrument for structuring the project.

A multitask project can be constructed from:

- A Master task (MAST)

The Master task is executed cyclically or periodically.

It forms the main section of the program and is executed sequentially.

- A Fast task (FAST)

The Fast task is executed periodically. It has a higher priority than the Master task. The Fast task is used for processes that are executed quickly and periodically.

- One to four AUX task(s))

The AUX tasks are executed periodically. They are used for slow processing and have the lowest priority.

The project can also be constructed with a single task. In this case, only the Master task is active.

Event Processing

Event processing takes place in event sections. Event sections are executed with higher priority than the sections of all other tasks. They are suited to processing that requires very short reaction times after an event is triggered.

The following section types are available for event processing:

- Sections for processing time controlled events (Timerx Section)
- Sections for processing hardware controlled events (Evtx Section)

The following programming languages are supported:

- FBD (Function Block Diagram)
- LD (Ladder Diagram Language)
- IL (Instruction List)
- ST (Structured Text)

Program Units

Program Units are autonomous programs in which the logic of the project is created.

The Program Units are executed in the order shown in the project browser (structural view).

A Program Unit is connected to a task. The same Program Unit cannot belong to more than one task at the same time.

The Program Unit includes:

- Public and local variables
- Sections

The following programming languages are supported:

- FBD (Function Block Diagram)
 - LD (Ladder Diagram Language)
 - SFC (Sequential Function Chart) only for sections in Program Unit which belongs to the MAST task
 - IL (Instruction List)
 - ST (Structured Text)
- Animation tables

Sections

Sections are autonomous programs in which the logic of the project is created.

The sections are executed in the order shown in the project browser (structural view).

A section is connected to a task. The same section cannot belong to more than one task at the same time.

The following programming languages are supported:

- FBD (Function Block Diagram)
- LD (Ladder Diagram Language)
- SFC (Sequential Function Chart) only for sections in MAST task
- IL (Instruction List)
- ST (Structured Text)

Subroutine

Subroutines are created as separate units in subroutine sections.

Subroutines are called from sections or from another subroutine.

Nesting of up to 8 levels is possible.

A subroutine cannot call itself (not recursive).

Subroutines are assigned a task. The same subroutine cannot be called by different tasks.

The following programming languages are supported:

- FBD (Function Block Diagram)
- LD (Ladder Diagram Language)
- IL (Instruction List)
- ST (Structured Text)

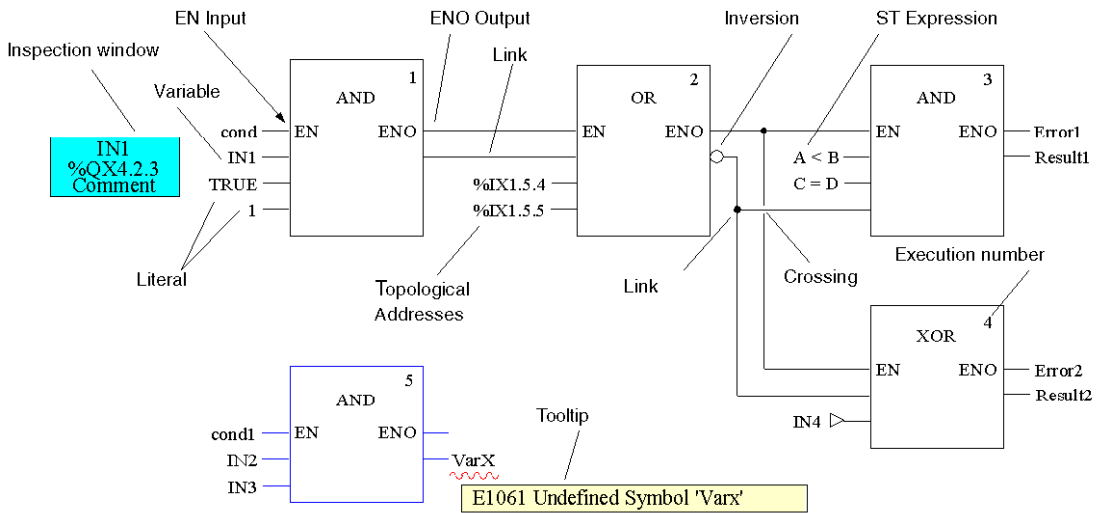
Function Block Diagram FBD

Introduction

The FBD editor is used for graphical function block programming according to IEC 61131-3.

Representation

Representation of an FBD section:



Objects

The objects of the FBD (Function Block Diagram) programming language help to divide a section into a number of:

- Elementary Functions (EFs),

- Elementary Function Blocks (EFBs)
- Derived Function Blocks (DFBs)
- Procedures
- Subroutine calls
- Jumps
- Links
- Actual Parameters
- Text objects to comment on the logic

Properties

FBD sections have a grid behind them. A grid unit consists of 10 coordinates. A grid unit is the smallest possible space between 2 objects in an FBD section.

The FBD programming language is not cell oriented but the objects are still aligned with the grid coordinates.

An FBD section can be configured in number of cells (horizontal grid coordinates and vertical grid coordinates).

The program can be entered using the mouse or the keyboard.

Input Aids

The FBD editor offers the following input aids:

- Toolbars for quick and easy access to the desired objects
- Syntax and semantics are checked as the program is being written.
 - Incorrect functions and function blocks are displayed in blue
 - Unknown words (e.g. undeclared variables) or unsuitable data types are marked with a red wavy line
 - Brief description of errors in the Quickinfo (Tooltip)
- Information for variables and pins can be displayed in a Quickinfo (Tooltip)
 - type, name, address and comment of a variable/expression
 - type, name and comment of an FFB pin
- Tabular display of FFBs
- Actual parameters can be entered and displayed as symbols or topological addresses
- Different zoom factors
- Tracking of links

- Optimization of link routes
- Display of inspection windows

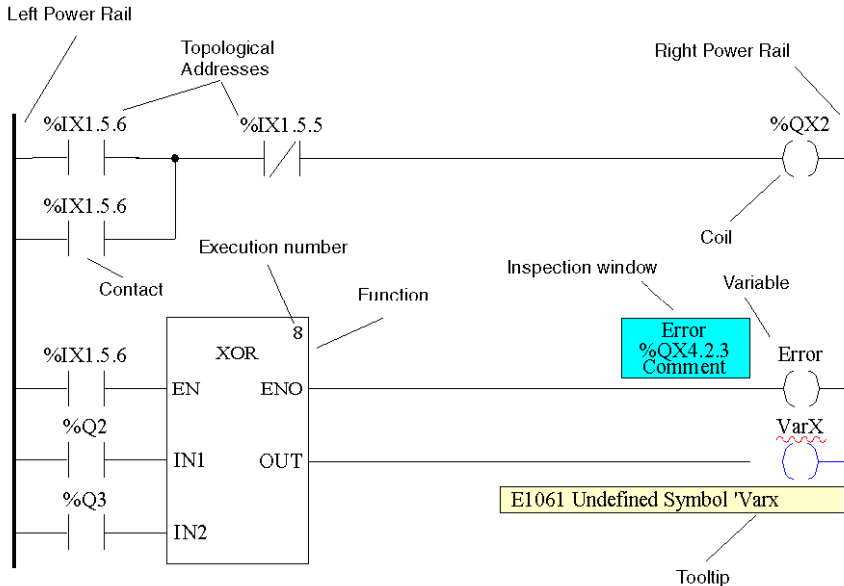
Ladder Diagram (LD) Language

Introduction

The LD editor is used for graphical ladder diagram programming according to IEC 61131-3.

Representation

Representation of an LD section:



Objects

The objects of the LD programming language help to divide a section into a number of:

- Contacts,
- Coils,
- Elementary Functions (EFs)
- Elementary Function Blocks (EFBs),

- Derived Function Blocks (DFBs)
- Procedures
- Control elements
- Operation and compare blocks which represent an extension to IEC 61131-3
- Subroutine calls
- Jumps
- Links
- Actual Parameters
- Text objects to comment on the logic

Properties

LD sections have a background grid that divides the section into lines and columns.

The LD programming language is cell oriented, i.e. only one object can be placed in each cell.

LD sections can be 11-63 columns and 17-3998 lines in size.

The program can be entered using the mouse or the keyboard.

Input Aids

The LD editor offers the following input aids:

- Objects can be selected from the toolbar, the menu or directly using shortcut keys
- Syntax and semantics are checked as the program is being written.
 - Incorrect objects are displayed in blue
 - Unknown words (e.g. undeclared variables) or unsuitable data types are marked with a red wavy line
 - Brief description of errors in the Quickinfo (Tooltip)
- Information for variables and for elements of an LD section, that can be connected to a variable (pins, contacts, coils, operation and compare blocks), can be displayed in a Quickinfo (Tooltip)
 - type, name, address and comment of a variable/expression
 - type, name and comment of FFB pins, contacts etc.
- Tabular display of FFBs
- Actual parameters can be entered and displayed as symbols or topological addresses
- Different zoom factors

- Tracking of FFB links
- Optimizing the link routes of FFB links
- Display of inspection windows

General Information about SFC Sequence Language

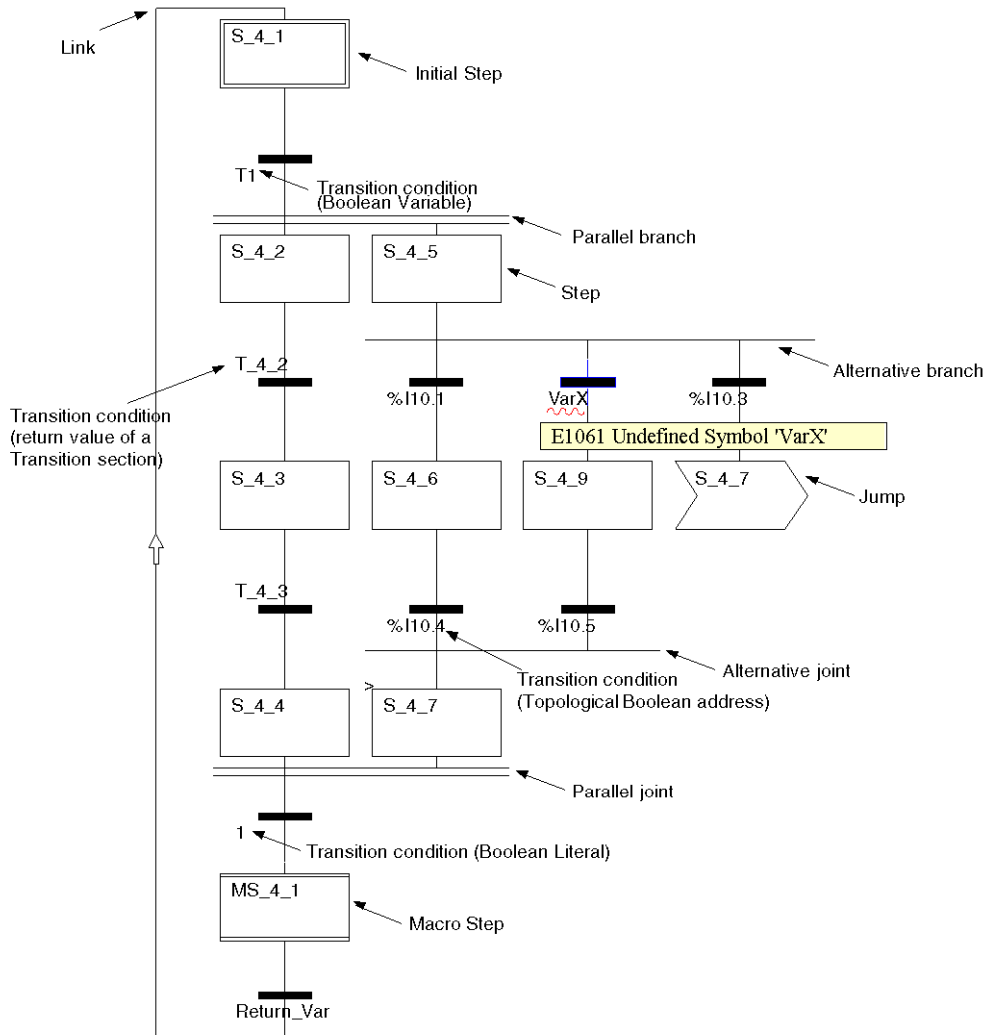
Introduction

The sequence language SFC (Sequential Function Chart), which conforms to IEC 61131-3, is described in this section.

IEC conformity restrictions can be lifted through explicit enable procedures. Features such as multi token, multiple initial steps, jumps to and from parallel strings etc. are then possible.

Representation

Representation of an SFC section:



Objects

An SFC section provides the following objects for creating a program:

- Steps
- Macro steps (embedded sub-step sequences)

- Transitions (transition conditions)
- Transition sections
- Action sections
- Jumps
- Links
- Alternative sequences
- Parallel sequences
- Text objects to comment on the logic

Properties

The SFC editor has a background grid that divides the section into 200 rows and 64 columns.

The program can be entered using the mouse or the keyboard.

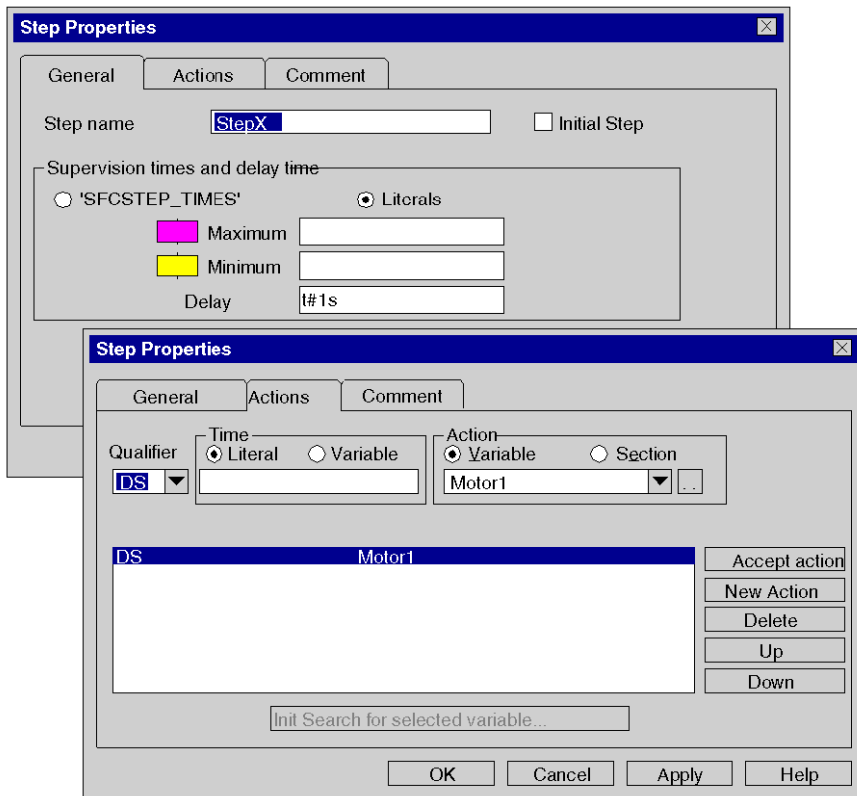
Input Aids

The SFC editor offers the following input aids:

- Toolbars for quick and easy access to the desired objects
- Automatic step numbering
- Direct access to actions and transition conditions
- Syntax and semantics are checked as the program is being written.
 - Incorrect objects are displayed in blue
 - Unknown words (e.g. undeclared variables) or unsuitable data types are marked with a red wavy line
 - Brief description of errors in the Quickinfo (Tooltip)
- Information for variables and for transitions can be displayed in a Quickinfo (Tooltip)
 - type, name, address and comment of a variable/expression
 - type, name and comment of transitions
- Different zoom factors
- Show/hide the allocated actions
- Tracking of links
- Optimization of link routes

Step Properties

Step properties:



The step properties are defined using a dialog box that offers the following features:

- Definition of initial steps
- Definition of diagnostics times
- Step comments
- Allocation of actions and their qualifiers

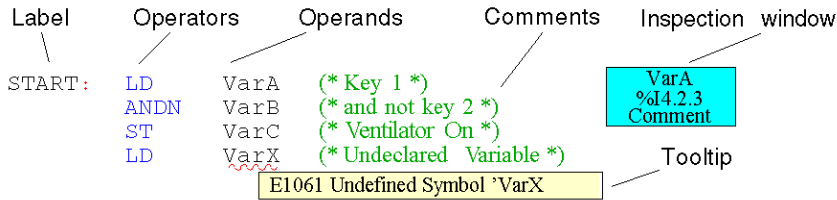
Instruction List IL

Introduction

The IL editor is used for instruction list programming according to IEC 61131-3.

Representation

Representation of an IL section:



Objects

An instruction list is composed of a series of instructions.

Each instruction begins on a new line and consists of:

- An operator
- A modifier if required
- One or more operands if required
- A label as a jump target if required
- A comment about the logic if required.

Input Aids

The IL editor offers the following input aids:

- Syntax and semantics are checked as the program is being written.
 - Keywords and comments are displayed in color
 - Unknown words (e.g. undeclared variables) or unsuitable data types are marked with a red wavy line
 - Brief description of errors in the Quickinfo (Tooltip)
- Tabular display of the functions and function blocks
- Input assistance for functions and function blocks
- Operands can be entered and displayed as symbols or topological addresses
- Display of inspection windows

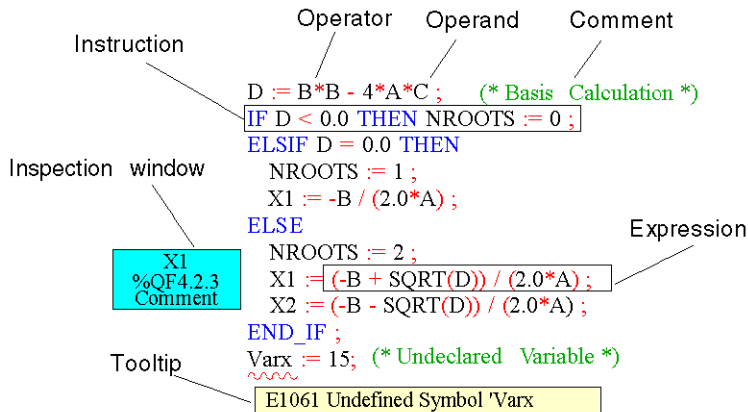
Structured Text ST

Introduction

The ST editor is used for programming in structured text according to IEC 61131-3.

Representation

Representation of an ST section:



Objects

The ST programming language works with "Expressions".

Expressions are constructions consisting of operators and operands that return a value when executed.

Operators are symbols representing the operations to be executed.

Operators are used for operands. Operands are variables, literals, function and function block inputs/outputs etc.

Instructions are used to structure and control the expressions.

Input Aids

The ST editor offers the following input aids:

- Syntax and semantics are checked as the program is being written.
 - Keywords and comments are displayed in color

- Unknown words (e.g. undeclared variables) or unsuitable data types are marked with a red wavy line
- Brief description of errors in the Quickinfo (Tooltip)
- Tabular display of the functions and function blocks
- Input assistance for functions and function blocks
- Operands can be entered and displayed as symbols or topological addresses
- Display of inspection windows

PLC Simulator

Introduction

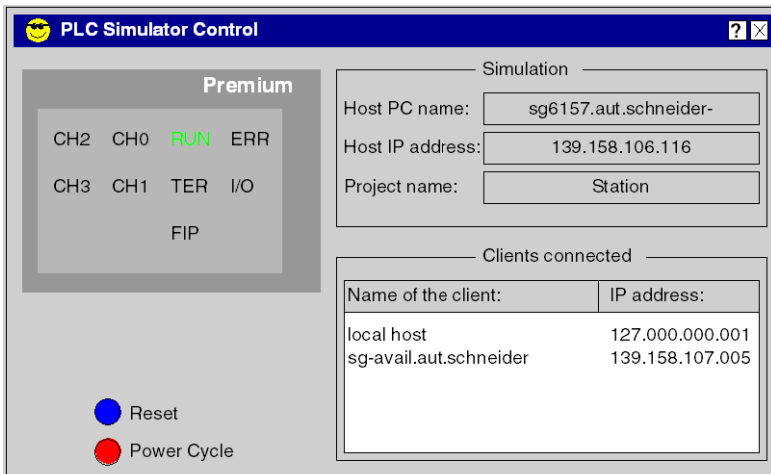
The PLC simulator enables error searches to be carried out in the project without being connected to a real PLC.

All project tasks (Mast, Fast, AUX and Event) that run on a real PLC are also available in the Simulator. The difference from a real PLC is the lack of I/O modules and communication networks (such as e.g. ETHWAY, Fipio and Modbus Plus) non-deterministic realtime behavior.

Naturally, all debugging functions, animation functions, breakpoints, forcing variables etc. are available with the PLC simulator.

Representation

Representation of a dialog box:



Structure of the Simulator

The simulator controller offers the following views:

- Type of simulated PLC
- Current status of the simulated PLC
- Name of the loaded project
- IP address and DNS name of the host PC for the simulator and all connected Client PCs
- Dialog box for simulating I/O events
- **Reset** button to reset the simulated PLC (simulated cold restart)
- **Power Off/On** button (to simulate a warm restart)
- Shortcut menu (right mouse button) for controlling the Simulator

Documentation

For detailed information refer to *EcoStruxure™ Control Expert, PLC Simulator*.

Export/Import

Introduction

The export and import functions allow you to use existing data in a new project. The XML export/import format makes it possible to provide or accept data from external software.

Export

The following objects can be exported:

- Complete projects, including configuration
- Program Units
- Sections of all programming languages
- Subroutine sections of all programming languages
- Derived function blocks (DFBs)
- Derived data types (DDTs)
- Device derived data types (Device DDTs)
- Variable declarations
- Operator Screen

Import

All objects that can be exported can naturally be imported as well.

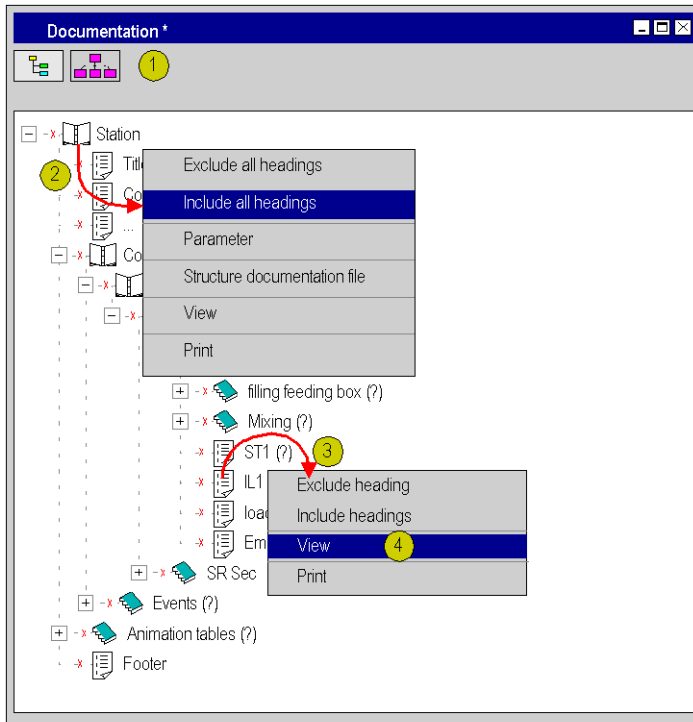
There are two types of import:

- **Direct import**
Imports the object exactly as it was exported.
- **Import with the assistant**
The assistant allows you to change the variables names, sections or functional modules. The mapping of addresses can also be modified.

User Documentation

User Documentation

Scope of the user documentation:



The following are just some of the services provided for documenting the project:

- Print the entire project (2) or in sections (3)
- Selection between structural and functional view (1)
- Adjustment of the result (footer, general information, etc.)
- Local printing for programming language editors, configurator, etc.
- Special indication (bold) for keywords
- Paper format can be selected
- Print preview (4)
- Documentation save

Debug Services

Searching for Errors in the User Application

The following are just some of the features provided to optimize debugging in the project:

- Set breakpoints in the programming language editors
- Step by step program execution, including step into, step out and step over
- Call memory for recalling the entire program path
- Control inputs and outputs

Online Mode

Online mode is when a connection is established between the PC and the PLC.

Online mode is used on the PLC for debugging, for animation and for changing the program.

A comparison between the project of the PC and project of the PLC takes place automatically when the connection is established.

This comparison can produce the following results:

- **Different projects on the PC and the PLC**

In this case, online mode is restricted. Only PLC control commands (e.g. start, stop), diagnostic services and variable monitoring are possible. Changes cannot be made to the PLC program logic or configuration. However, the downloading and uploading functions are possible and run in an unrestricted mode (same project on PC and PLC).

- **Same projects on the PC and the PLC**

There are two different possibilities:

- **ONLINE SAME, BUILT**

The last project generation on the PC was downloaded to the PLC and no changes were made afterwards, i.e. the projects on the PC and the PLC are absolutely identical.

In this case, all animation functions are available and unrestricted.

- **ONLINE EQUAL, NOT BUILT**

The last project generation on the PC was downloaded to the PLC, however changes were made afterwards.

In this case, the animation functions are only available in the unchanged project components.

Animation

Different possibilities are provided for the animation of variables:

- **Section animation**

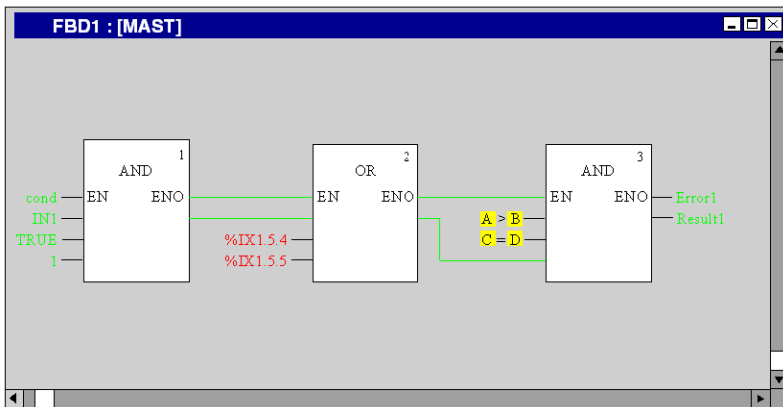
All programming languages (FBD, LD, SFC, IL and ST) can be animated.

The variables and connections are animated directly in the section.

```

ST1 : [MAST]
    TIMER(IN := NOT pulse;
          PT := T#1s; (* Blink timer *))
    pulse := TIMER.Q;

    (* Count every pulse *)
    IF pulse = 1 THEN
        count := count + 1;
    END_IF;
    (* Animate lights according to counter *)
    CASE count OF
        1: out1 := TRUE;
        2: out2 := TRUE;
    ELSE (* All lights are on, switch then off again and start from
        out1 := FALSE;
        out2 := FALSE;
    
```



- **Tooltips**

A tooltip with the value of a variable is displayed when the mouse pointer passes over that variable.

```

ST1 : [MAST]
TIMER(IN := NOT pulse;
      ET := t#1s; (* Blink timer *))
pulse := TIMER.Q;

(* Count every pulse *)
IF pulse = 1 THEN
  count := count + 1;
END_IF;
66
(* Animate lights according to counter *)
CASE count OF
  1: out1 := TRUE;
  2: out2 := TRUE;
ELSE (* All lights are on, switch then off again and start from
      out1 := FALSE;
      out2 := FALSE;

```

- **Inspection window**

An inspection window can be created for any variable. This window displays the value of the variable, the address and any comments (if available). This function is available in all programming languages.

```

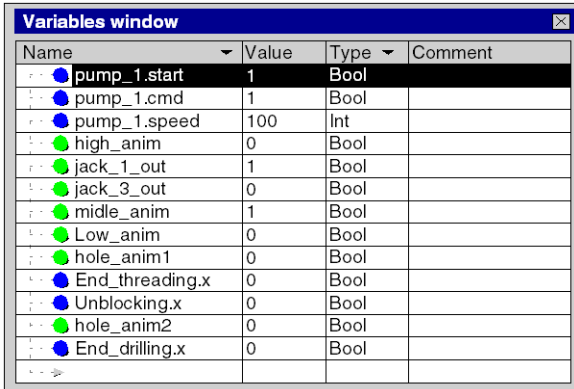
ST1 : [MAST]
TIMER(IN := NOT pulse;
      ET := t#1s; (* Blink timer *))
pulse := TIMER.Q;

(* Count every pulse *)
IF pulse = 1 THEN
  count := count + 1;
END_IF;
7
(* Animate lights according to counter *)
CASE count OF
  1: out1 := TRUE;
  2: out2 := TRUE;
ELSE (* All lights are on, switch then off again and start from
      out1 := FALSE;
      out2 := FALSE;

```

- **Variables window**

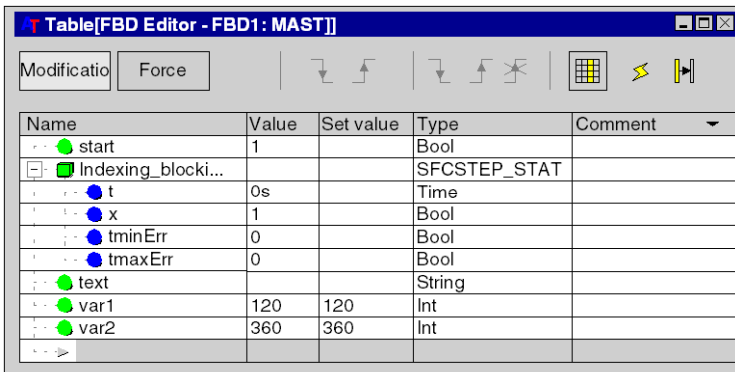
This window displays all variables used in the current section.



| Name | Value | Type | Comment |
|-----------------|-------|------|---------|
| pump_1.start | 1 | Bool | |
| pump_1.cmd | 1 | Bool | |
| pump_1.speed | 100 | Int | |
| high_anim | 0 | Bool | |
| jack_1_out | 1 | Bool | |
| jack_3_out | 0 | Bool | |
| midle_anim | 1 | Bool | |
| Low_anim | 0 | Bool | |
| hole_anim1 | 0 | Bool | |
| End_threading.x | 0 | Bool | |
| Unblocking.x | 0 | Bool | |
| hole_anim2 | 0 | Bool | |
| End_drilling.x | 0 | Bool | |

- **Animation table**

The value of all variables in the project can be displayed, changed or forced in animation tables. Values can be changed individually or simultaneously together.



| Name | Value | Set value | Type | Comment |
|--------------------|-------|-----------|--------------|---------|
| start | 1 | | Bool | |
| Indexing_blocki... | | | SFCSTEP_STAT | |
| t | 0s | | Time | |
| x | 1 | | Bool | |
| tminErr | 0 | | Bool | |
| tmaxErr | 0 | | Bool | |
| text | | | String | |
| var1 | 120 | 120 | Int | |
| var2 | 360 | 360 | Int | |

Watch Point

Watch points allow you to view PLC data at the exact moment at which it is created (1) and not only at the end of a cycle.

Animation tables can be synchronized with the watch point (2).

A counter (3) determines how often the watch point has been updated.

ST section with watch point:

The screenshot shows three windows from a software development environment:

- Watch Point:** A dialog box with a value of 341 and a circled '3' next to it.
- ST (Section): My_ST [MAST]:** A code editor window containing the following code:

```
if pump_1.start
  then pump_1.cmd:= true;
  else pump_1.cmd:= false; pump_1.speed:= 0;
end_if;
if pump_1.cmd then pump_1.speed:= pump_1.speed + 1; end_if;
if pump_1.speed>100 then pump_1.speed:= 100; end_if;
(* animation drilling & flureadinf *)
high_anim:= not jack_1_out and not jack_3_out;
```

Red arrows point from circled '1' to the first 'if' statement and from circled '2' to the second 'if' statement.
- Table[FBD-Editor - My_ST: MAST]:** A table with the following data:

| Name | Value | Type | Comment |
|--------------------|-------|----------|---------|
| start | | Bool | |
| Indexing_blocki... | | SFCST... | |
| t | | Time | |
| x | | Bool | |
| tminErr | | Bool | |

Breakpoint

Breakpoints allow you to stop processing of the project at any point.

ST section with breakpoint:

The screenshot shows a code editor window titled "ST1: [MAST]" with the following code:

```
TIMER(IN := NOT pulse;
      PT := t#1s; (* Blink timer *))
pulse := TIMER.Q;

(* Count every pulse *)
IF pulse = 1 THEN
  count := count + 1;
END_IF;

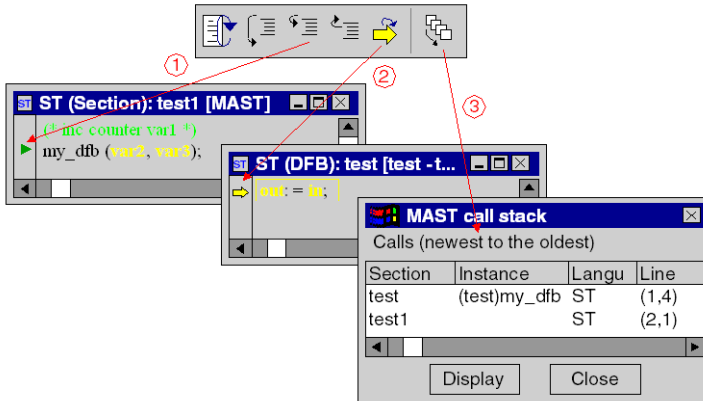
(* Animate lights according to counter *)
CASE count OF
  1: out1 := TRUE;
  2: out2 := TRUE;
ELSE (* All lights are on, switch then off again and start from
      out1 := FALSE;
      out2 := FALSE;
```

A red circle labeled "Breakpoint" is positioned on the left margin, pointing to the line "IF pulse = 1 THEN".

Single Step Mode

Single step mode allows you to execute the program step by step. Single step functions are provided if the project was stopped by reaching a breakpoint or if it is already in single step mode.

ST section in single step mode:



The following functions are provided in single step mode:

- Step by step execution of the program
- StepIn (1)
- StepOut
- StepOver
- Show Current Step (2)
- Call memory (3)

When the "step into" function is executed several times, the call memory enables the display of the entire path, starting with the first breakpoint

NOTE: Running the PLC program in step by step mode, as well as entering (StepIn) in a read/write protected section may lead to the inability to read the program and exit from the section. The user must switch the PLC in "Stop" mode to get back to the initial state.

Bookmarks

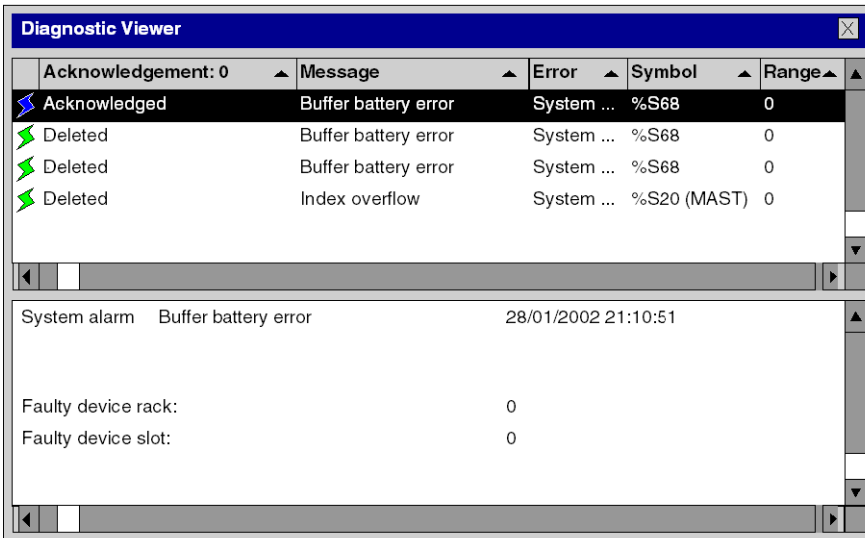
Bookmarks allow you to select code sections and easily find them again.

Diagnostic Viewer

Description

Control Expert provides system and project diagnostics.

Errors which occur are displayed in a diagnostics window. The section which caused the error can be opened directly from the diagnostics window in order to correct the error.



The screenshot shows the 'Diagnostic Viewer' window. It contains a table of error messages and a detailed view of a selected error.

| Acknowledgement: 0 | Message | Error | Symbol | Range |
|--------------------|----------------------|------------|-------------|-------|
| ✓ Acknowledged | Buffer battery error | System ... | %S68 | 0 |
| ✓ Deleted | Buffer battery error | System ... | %S68 | 0 |
| ✓ Deleted | Buffer battery error | System ... | %S68 | 0 |
| ✓ Deleted | Index overflow | System ... | %S20 (MAST) | 0 |

| | | |
|---------------------|----------------------|---------------------|
| System alarm | Buffer battery error | 28/01/2002 21:10:51 |
| Faulty device rack: | 0 | |
| Faulty device slot: | 0 | |

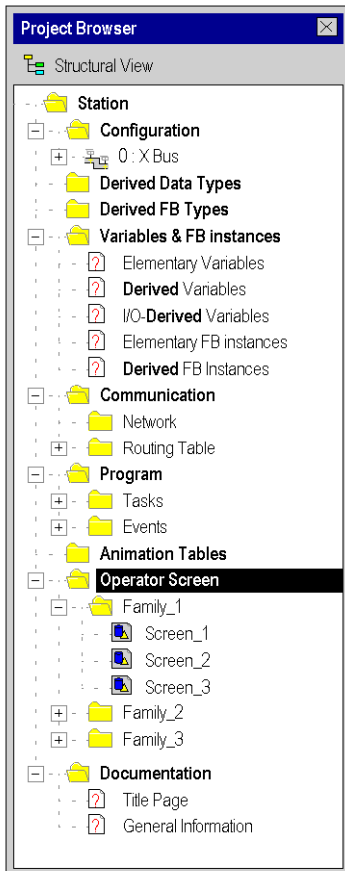
Operator Screen

Introduction

Operator windows visualize the automation process.

The operator screen editor makes it easy to create, change and manage operator screens.

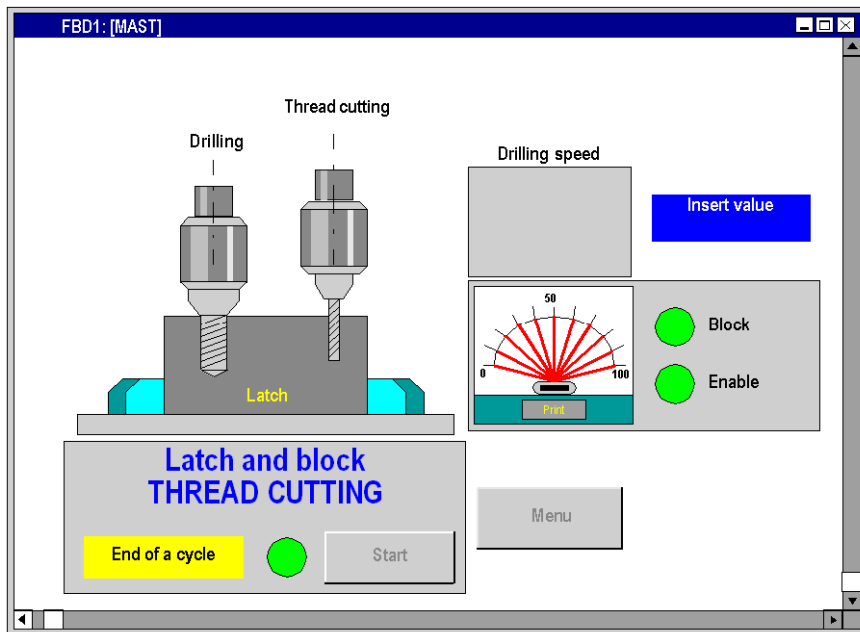
Operator screens are created and accessed via the project browser.



Operator Screen Editor

An operator window contains much information (dynamic variables, overviews, written text, etc.) and makes it easy to monitor and change automation variables.

Operator Screen



The operator screen editor offers the following features:

- Extensive visualization functions
 - Geometric elements
 - Line, rectangle, ellipse, curve, polygon, bitmap, text
 - Control elements
 - Buttons, control box, shifter, screen navigation, hyperlinks, input field, rotating field
 - Animation elements
 - Bar chart, trend diagram, dialog, date, disappear, blinking colors, variable animation
- Create a library for managing graphical objects
- Copying objects
- Creating a list of all variables used in the operator screen
- Creating messages to be used in the operator screen
- Direct access from the operator screen to the animation table or the cross reference table for one or more variables
- Tooltips give additional information about the variables
- Managing operator screens in families

- Import/export of individual operator screens or entire families

Application Structure

What's in This Part

| | |
|---|-----|
| Description of the Available Functions for Each Type of PLC | 80 |
| Application Program Structure | 85 |
| Application Memory Structure | 119 |
| Operating Modes | 135 |

In This Part

This part describes the application program and memory structures associated with each type of PLC.

Description of the Available Functions for Each Type of PLC

What’s in This Chapter

Functions Available for the Different Types of PLC..... 80

Subject of this Chapter

This chapter lists the available functions by PLC platform and processor type.

They essentially concern useful features for programming. Other functions are described in the installation manuals for each individual type of PLC.

Functions Available for the Different Types of PLC

Programming Languages

The following languages are available for platforms Modicon M580, Modicon M340, Momentum, Premium, Atrium, and Quantum:

- LD
- FBD
- ST
- IL
- SFC

NOTE: Only LD and FBD languages are available on Modicon M580 Safety and Modicon Quantum Safety CPUs.

Tasks and Processes

The following tables describe the available tasks and processes for Premium and Atrium:

| Tasks Processes | Premium: TSX Processors | | | Atrium: TSX Processors |
|--|-------------------------|---|---------------------|--------------------------|
| | P57 0244 P57 1** | P57 2** P57 3** P57 4** H57 24M H57 44M | P57 5** P57 6634 | PCI 57 204 PCI 57 354 |
| Master task cyclic or periodic | X | X | X | X |
| Fast task periodic | X | X | X | X |
| Auxiliary tasks periodic | - | - | 4 | - |
| Program Unit | - | - | - | - |
| Maximum size of a section | 64 Kb | | | 64 Kb |
| I/O type event processing | 32 | 64 | 128 | 64 |
| Timer type event processing | - | - | 32 | - |
| Total of I/O type and Timer type event processing | 32 | 64 | 128 | 64 |
| <p>X or Value: Available tasks or processes (Value is the maximum number).</p> <p>-: Unavailable tasks or processes.</p> <p>(1) Depends on the available processor memory.</p> | | | | |

The following tables describe the available tasks and processes for Quantum:

| Tasks Processes | Quantum: 140 CPU Processors | | |
|---|-----------------------------|--------|---------|
| | 31• ... | 651•• | 651 60S |
| | 43• ... | 652 60 | 671 60S |
| | 53• ... | 670 60 | |
| | | 671 60 | |
| | | 672 60 | |
| | | 672 61 | |
| Master task cyclic or periodic | X | X | X |
| Fast task periodic | X | X | - |
| Auxiliary tasks periodic | - | 4 | - |
| Program Unit | - | - | - |
| Maximum size of a section | 64 Kb | (1) | - |
| I/O type event processing | 64 | 128 | - |
| Timer type event processing | 16 | 32 | - |
| Total of I/O type and Timer type event processing | 64 | 128 | - |
| <p>X or Value: Available tasks or processes (Value is the maximum number).</p> <p>-: Unavailable tasks or processes.</p> <p>(1) Depends on the available processor memory.</p> | | | |

The following tables describe the available tasks and processes for M340:

| Tasks Processes | Modicon M340 Processors | |
|-----------------------------------|-------------------------|----------|
| | P34 1000 | P34 20•• |
| Master task cyclic or periodic | X | X |
| Fast task periodic | X | X |
| Auxiliary tasks periodic | - | - |

| Tasks Processes | Modicon M340 Processors | |
|---|-------------------------|----------|
| | P34 1000 | P34 20•• |
| Program Unit | X | X |
| Maximum size of a section | (1) | |
| I/O type event processing | 32 | 64 |
| Timer type event processing | 16 | 32 |
| Total of I/O type and Timer type event processing | 32 | 64 |
| <p>X or Value: Available tasks or processes (Value is the maximum number). -: Unavailable tasks or processes. (1) Depends on the available processor memory.</p> | | |

The following tables describe the available tasks and processes for M580:

| Tasks Processes | Modicon M580 BME Processors | | | | |
|-----------------------------------|-----------------------------|--|----------------------------------|------------------------|-------------------------------------|
| | P58 1020 P58 20•0 | P58 30•0 P58 40•0 P58 5040 P58 6040 | H58 2040 H58 4040 H58 6040 | P58 2040S P58 4040S | H58 2040S H58 4040S H58 6040S |
| Master task cyclic or periodic | X | X | X | X (2) | X (2) |
| Fast task periodic | X | X | X | X | X |
| Auxiliary tasks periodic | 2 | 2 | - | 2 | - |
| Program Unit | X | X | X | X | X |
| Maximum size of a section | (1) | (1) | (1) | (1) | (1) |
| I/O type event processing | 64 | 128 | - | 128 | - |
| Timer type event processing | 32 | 32 | - | 32 | - |

| Tasks Processes | Modicon M580 BME Processors | | | | |
|---|-----------------------------|--|----------------------------------|------------------------|-------------------------------------|
| | P58 1020 P58 20•0 | P58 30•0 P58 40•0 P58 5040 P58 6040 | H58 2040 H58 4040 H58 6040 | P58 2040S P58 4040S | H58 2040S H58 4040S H58 6040S |
| Total of I/O type and Timer type event processing | 64 | 128 | - | 128 | - |
| <p>X or Value: Available tasks or processes (Value is the maximum number).</p> <p>-: Unavailable tasks or processes.</p> <p>(1) Depends on the available processor memory.</p> <p>(2) + a dedicated SAFE task.</p> | | | | | |

The following tables describe the available tasks and processes for Momentum:

| Tasks Processes | Momentum Processors |
|---|--------------------------------|
| | 171 CBU 78090 171 CBU 9809• |
| Master task cyclic or periodic | X |
| Fast task periodic | - |
| Auxiliary tasks periodic | - |
| Program Unit | - |
| Maximum size of a section | (1) |
| I/O type event processing | - |
| Timer type event processing | - |
| Total of I/O type and Timer type event processing | - |
| <p>X or Value: Available tasks or processes (Value is the maximum number).</p> <p>-: Unavailable tasks or processes.</p> <p>(1) Depends on the available processor memory.</p> <p>(2) + a dedicated SAFE task.</p> | |

Application Program Structure

What's in This Chapter

| | |
|--|-----|
| Description of Tasks and Processes | 85 |
| Description of Program Units | 90 |
| Description of Sections and Subroutines..... | 92 |
| Mono Task Execution | 96 |
| Multitasking Execution..... | 102 |

Subject of this Chapter

This chapter describes the structure and execution of the programs created using the Control Expert software.

Description of Tasks and Processes

Subject of this Section

This section describes the tasks and processes that comprise the application program.

Presentation of the Master Task

General

The master task represents the main task of the application program. It is obligatory and created by default.

Structure

The master task (MAST) is made up of Program Units and/or sections and subroutines.

NOTE: Program Units are only available for Modicon M580 and M340.

Each section of the master task is programmed in the following languages: LD, FBD, IL, ST or SFC.

The subroutines are programmed in LD, FBD, IL, or ST and are called in the task sections.

NOTE: SFC can be used only in the master task sections. The number of sections programmed in SFC is unlimited.

Execution

You can choose the type of master task execution:

- cyclic (default selection)
- or periodic (1 to 255 ms)

Control

The master task can be controlled by program, by bits and system words.

| System objects | Description |
|----------------|--|
| %SW0 | Task period. |
| %S30 | Master task activation. |
| %S11 | Watchdog error. |
| %S19 | Period overrun. |
| %SW27 | Number of ms spent in the system during the last Mast cycle. |
| %SW28 | Maximum overhead time (in ms). |
| %SW29 | Minimum overhead time (in ms). |
| %SW30 | Execution time (in ms) of the last cycle. |
| %SW31 | Execution time (in ms) of the longest cycle. |
| %SW32 | Execution time (in ms) of the shortest cycle. |

Presentation of the Fast Task

General

The fast task is intended for short duration and periodic processing tasks.

Structure

The fast task (FAST) is made up of Program Units and/or sections and subroutines.

NOTE: Program Units are only available for Modicon M580 and M340.

Each section of the fast task is programmed in one of the following languages: LD, FBD, IL or ST.

SFC language cannot be used in the sections of a fast task.

Subroutines are programmed in LD, FBD, IL, or ST language and are called in the task sections.

Execution

The execution of the fast task is periodic.

It is higher priority than the master task.

The period of the fast task (FAST) is fixed by configuration, from 1 to 255 ms.

The executed program must however remain short to avoid the overflow of lower-priority tasks.

Control

The fast task can be controlled by program by bits and system words.

| System objects | Description |
|----------------|---|
| %SW1 | Task period. |
| %S31 | Fast task activation. |
| %S11 | Watchdog error |
| %S19 | Period overrun. |
| %SW33 | Execution time (in ms) of the last cycle. |
| %SW34 | Execution time (in ms) of the longest cycle. |
| %SW35 | Execution time (in ms) of the shortest cycle. |

Presentation of Auxiliary Tasks

General

The auxiliary tasks are intended for slower processing tasks. These are the least priority tasks.

It is possible to program up to four auxiliary tasks (AUX0, AUX1, AUX2 or AUX3) on the Premium TSX P57 5•• and Quantum 140 CPU 6•••• PLCs.

It is possible to program up to two auxiliary tasks (AUX0, AUX1) on the Modicon M580 BME P58 •••• PLCs.

Auxiliary tasks are not available for Modicon M340 PLCs.

Structure

The auxiliary tasks (AUX) are made up of Program Units and/or sections and subroutines.

NOTE: Program Units are only available for Modicon M580 and M340.

Each section of the auxiliary task is programmed in one of the following languages: LD, FBD, IL or ST.

The SFC language is not usable in the sections of an auxiliary task.

A maximum of 64 subroutines can be programmed in the LD, FBD, IL or ST language. These are called in the task sections.

Execution

The execution of auxiliary tasks is periodic.

They are the least priority.

The auxiliary task period can be fixed from 10 ms to 2550 ms.

Control

The auxiliary tasks can be controlled by program by system bits and words:

| System objects | Description |
|----------------|--|
| %SW2 | Period of auxiliary task 0 |
| %SW3 | Period of auxiliary task 1 |
| %SW4 | Period of auxiliary task 2 |
| %SW5 | Period of auxiliary task 3 |
| %S32 | Activation of auxiliary task 0 |
| %S33 | Activation of auxiliary task 1 |
| %S34 | Activation of auxiliary task 2 |
| %S35 | Activation of auxiliary task 3 |
| %S11 | Watchdog error |
| %S19 | Period overrun. |
| %SW36 | Execution time (in ms) of the last cycle of auxiliary task 0 |
| %SW39 | Execution time (in ms) of the last cycle of auxiliary task 1 |
| %SW42 | Execution time (in ms) of the last cycle of auxiliary task 2 |

| System objects | Description |
|----------------|--|
| %SW45 | Execution time (in ms) of the last cycle of auxiliary task 3 |
| %SW37 | Execution time (in ms) of the longest cycle of auxiliary task 0 |
| %SW40 | Execution time (in ms) of the longest cycle of auxiliary task 1 |
| %SW43 | Execution time (in ms) of the longest cycle of auxiliary task 2 |
| %SW46 | Execution time (in ms) of the longest cycle of auxiliary task 3 |
| %SW38 | Execution time (in ms) of the shortest cycle of auxiliary task 0 |
| %SW41 | Execution time (in ms) of the shortest cycle of auxiliary task 1 |
| %SW44 | Execution time (in ms) of the shortest cycle of auxiliary task 2 |
| %SW47 | Execution time (in ms) of the shortest cycle of auxiliary task 3 |

Overview of Event Processing

General

Event processing is used to reduce the response time of the application program to events:

- coming from input/output modules,
- from event timers.

These processing tasks are performed with priority over all other tasks. They are therefore suited to processing tasks requiring a very short response time in relation to the event.

The number of event processing tasks, page 80 that can be programmed depends on the type of processor.

Structure

An event processing task is monosectional, and made up of a single (unconditioned) section.

It is programmed in either LD, FBD, IL or ST language.

Two types of event are offered:

- I/O event: for events coming from input/output modules
- TIMER event: for events coming from event timers.

Execution

The execution of an event processing task is asynchronous.

The occurrence of an event reroutes the application program to the processing task associated with the input/output channel or event timer which caused the event.

Control

The following system bits and words can be used to control event processing tasks during the execution of the program.

| System objects | Description |
|----------------|---|
| %S38 | Activation of event processing. |
| %S39 | Saturation of the event call management stack. |
| %SW48 | Number of IO events and telegram processing tasks executed. NOTE: TELEGRAM is available only for PREMIUM. |
| %SW75 | Number of timer type events in the queue. |

Description of Program Units

Aim of this Section

This section describes the Program Units that make up a task.

Description of Program Units

Overview of the Program Unit

Program Units are autonomous programming entities (only available for Modicon M580 and M340).

The Program Unit includes:

- Public and local variables
- Sections

The following programming languages are supported:

- FBD (Function Block Diagram)
- LD (Ladder Diagram Language)

- SFC (Sequential Function Chart) only for sections in Program Unit which belongs to the MAST task
- IL (Instruction List)
- ST (Structured Text)
- Animation tables

The Program Units are linked to a task. The same Program Unit cannot belong simultaneously to several tasks.

The sections and Program Units under a task are executed in the order of their programming in the browser window (structure view).

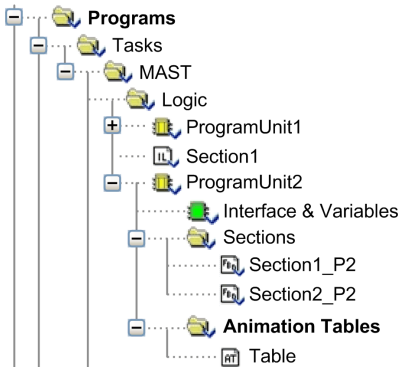
The sections under a Program Unit are executed in the order of their programming in the browser window (structure view).

The identification tags of the instruction lines, the contact networks, and so on, are specific to each section of the Program Unit (no program jump to another section of the same Program Unit is possible).

An execution condition can be associated with one or more Program Units and or sections of Program Units.

Example

The following diagram shows a task structured into Program Units and sections:



The execution order in this MAST task example, starts with ProgramUnit1, continues with Section1 and ends with ProgramUnit2. Inside ProgramUnit2 the execution order is first Section1_P2 then Section2_P1.

Characteristics of a Program Unit

The following table describes the characteristics of a Program Unit:

| Characteristic | Description |
|-------------------------|---|
| Name | 32 characters maximum (accents are possible, but spaces are not allowed). |
| Section Language | LD, FBD, IL, ST or SFC |
| Task or processing | Master, fast, auxiliary |
| Condition (optional) | A BOOL or EBOOL type bit variable can be used to condition the execution of the Program Unit. |
| Comment | 256 characters maximum |
| Protection | Write-protection, read/write protection. |

Description of Sections and Subroutines

Aim of this Section

This section describes the sections and the subroutines that make up a task and a Program Unit.

Description of Sections

Overview of the Sections

Sections are autonomous programming entities.

The identification tags of the instruction lines, the contact networks, etc. are specific to each section (no program jump to another section is possible).

These are programmed either in:

- Ladder language (LD)
- Functional block language (FBD)
- Instruction List (IL)
- Structured Text (ST)
- or Sequential Function Charting (SFC)

on condition that the language is accepted in the task.

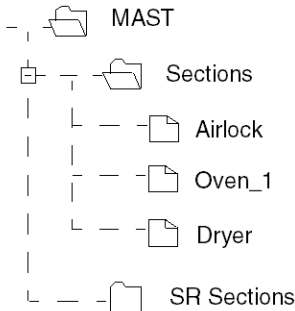
The sections are executed in the order of their programming in the browser window (structure view).

An execution condition can be associated with one or more sections in the master, fast and auxiliary tasks, but not in the event processing tasks.

The sections are linked to a task. The same section cannot belong simultaneously to several tasks.

Example

The following diagram shows a task structured into sections.



Characteristics of a Section

The following table describes the characteristics of a section.

| Characteristic | Description |
|----------------------|--|
| Name | 32 characters maximum (accents are possible, but spaces are not allowed). |
| Language | LD, FBD, IL, ST or SFC |
| Task or processing | Master, fast, auxiliary, event |
| Condition (optional) | A BOOL or EBOOL type bit variable can be used to condition the execution of the section. |
| Comment | 256 characters maximum |
| Protection | Write-protection, read/write protection. |

Description of SFC sections

General

The sections in Sequential Function Chart language are made up of:

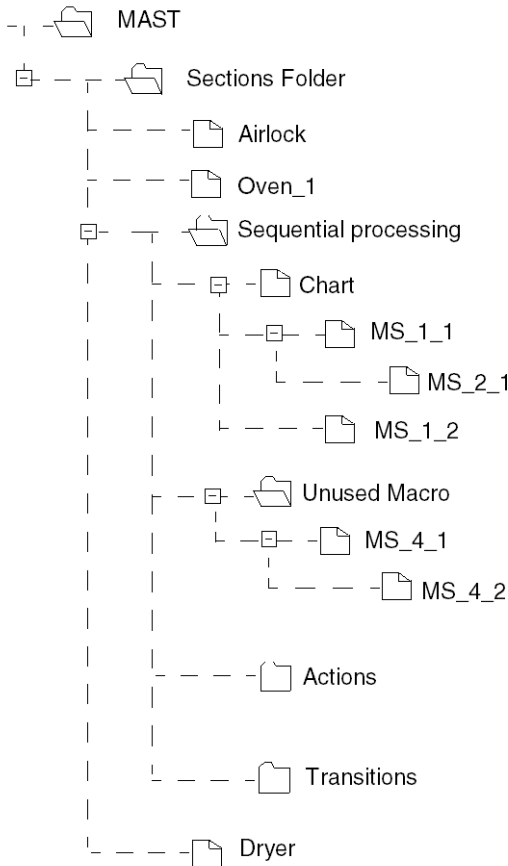
- a main chart programmed in SFC

- macro steps (MS) programmed in SFC
- actions and transitions programmed in LD, FBD, ST, or IL

The SFC sections are programmable only in the master task (see detailed description of SFC sections)

Example

The following diagram gives an example of the structure of an SFC section, and uses the chart to show the macro step calls that are used.



Description of Subroutines

Overview of Subroutines

Subroutines are programmed as separate entities, either in:

- Ladder language (LD),
- Functional block language (FBD),
- Instruction List (IL),
- Structured Text (ST).

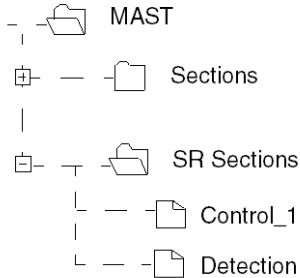
The calls to subroutines are carried out in the sections or from another subroutine.

A subroutine cannot call itself (non-recursive).

Subroutines are also linked to a task. The same subroutine cannot be called from several different tasks.

Example

The following diagram shows a task structured into sections and subroutines.



Characteristics of a Subroutine

The following table describes the characteristics of a subroutine.

| Characteristic | Description |
|----------------|---|
| Name | 32 characters maximum (accents are possible, but spaces are not allowed). |
| Language | LD, FBD, IL, or ST. |
| Task | MAST, FAST or Auxiliary |
| Comment | 1024 characters maximum |

Mono Task Execution

Subject of this Section

This section describes how a mono task application operates.

Description of the Master Task Cycle

General

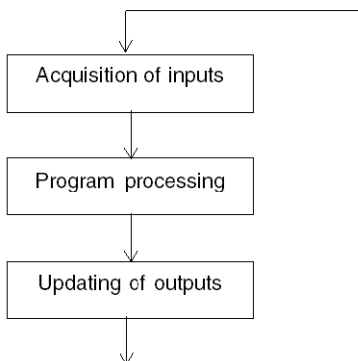
The program for a mono task application is associated with a single user task, the master task, page 85.

You can choose the type of master task execution:

- cyclic
- periodic

Illustration

The following illustration shows the operating cycle.



Description of the Different Phases

The table below describes the operating phases.

| Phase | Description |
|-----------------------|--|
| Acquisition of inputs | Writing to memory of the status of the data on the inputs of the discrete and application-specific modules associated with the task, These values can be modified by forcing values. |
| Program processing | Execution of application program, written by the user, |
| Updating of outputs | Writing of output bits or words to the discrete or application-specific modules associated with the task depending on the state defined by the application. As for the inputs, the values written to the outputs can be modified by forcing values. |

NOTE: During the input acquisition and output update phases, the system also implicitly monitors the PLC (management of system bits and words, updating of current values of the real time clock, updating of status LEDs and LCD screens (not for Modicon M340), detection of changes between RUN/STOP, etc.) and the processing of requests from the terminal (modifications and animation).

Operating Mode

PLC in RUN, the processor carries out internal processing, input acquisition, processing of the application program and the updating of outputs in that order.

PLC in STOP, the processor carries out:

- internal processing,
- input acquisition (1),
- and depending on the chosen configuration:
 - fallback mode: the outputs are set to fallback position.
 - maintain mode: the last value of the outputs is maintained.

(1) For Quantum PLCs, input acquisition is inhibited when the PLC is in STOP.

NOTE: For information about inhibiting and activating tasks using system bits refer to Task Control, page 107.

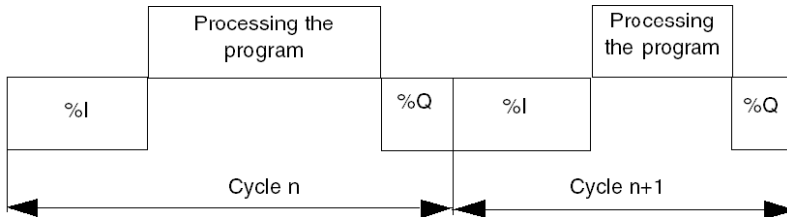
Mono Task: Cyclic Execution

General

The master task operates as outlined below. A description is provided of cyclic execution of the master task in mono task operation.

Operation

The following drawing shows the execution phases of the PLC cycle.



%I Reading of inputs

%Q Writing of outputs

Description

This type of operation consists of sequencing the task cycles, one after another.

After having updated the outputs, the system performs its own specific processing then starts another task cycle, without pausing.

Cycle Check

The cycle is checked by the watchdog, page 99.

Periodic Execution

Description

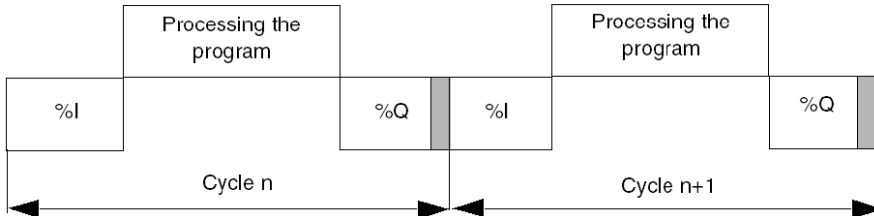
In this operating mode, input acquisition, the processing of the application program and the updating of outputs are all carried out periodically over a defined period of 1 to 255 ms.

At the start of the PLC cycle, a time out whose current value is initialized to the defined period starts the countdown.

The PLC cycle must be completed before this time out expires and launches a new cycle.

Operation

The following diagram shows the execution phases of the PLC cycle.



%I Reading of inputs

%Q Writing of outputs

Operating Mode

The processor carries out internal processing, input acquisition, processing of the application program and the updating of outputs in that order.

- If the period is not yet over, the processor completes its operating cycle until the end of the period by performing internal processing.
- If the operating time is longer than that assigned to the period, the PLC signals a period overrun by setting the system bit **%S19** of the task to 1. Processing then continues and is executed fully (however, it must not exceed the watchdog time limit). The following cycle is started after the outputs have been implicitly written for the current cycle.

Cycle Check

Two checks are carried out:

- period overrun, page 99,
- by watchdog, page 99.

Control of Cycle Time

General

The period of master task execution, in cyclic or periodic operation, is controlled by the PLC (watchdog) and must not exceed the value defined in Tmax configuration (1500 ms by default, 1.5 s maximum).

Software Watchdog (Periodic or Cyclic Operation)

If watchdog overflow should occur, the application is declared in error, which causes the PLC to stop immediately (HALT state).

The bit %S11 indicates a watchdog overflow. It is set to 1 by the system when the cycle time becomes greater than the watchdog.

The word %SW11 contains the watchdog value in ms. This value is not modifiable by the program.

NOTE:

- The reactivation of the task requires the terminal to be connected in order to analyze the cause of the error, correct it, reinitialize the PLC and switch it to RUN.
- It is not possible to exit HALT by switching to STOP. To do this you must reinitialize the application to ensure consistency of data.

Control in Periodic Operation

In periodic operation, an additional control enables a period overrun to be detected. A period overrun does not cause the PLC to stop if it remains less than the watchdog value.

The bit %S19 indicates a period overflow. It is set to 1 by the system, when the cycle time becomes greater than the task period.

The word %SW0 contains the value of the period (in ms). It is initialized on cold restart by the defined value. It can be changed by the user.

Exploitation of Master Task Execution Times

The following system words can be used to obtain information on the cycle time:

- %SW30 contains the execution time of the last cycle
- %SW31 contains the execution time of the longest cycle
- %SW32 contains the execution time of the shortest cycle

NOTE: These different items of information can also be accessed explicitly from the configuration editor.

Execution of Quantum Sections with Remote Inputs/Outputs

General

Quantum PLCs have a specific section management system. It applies to stations with remote inputs/outputs.

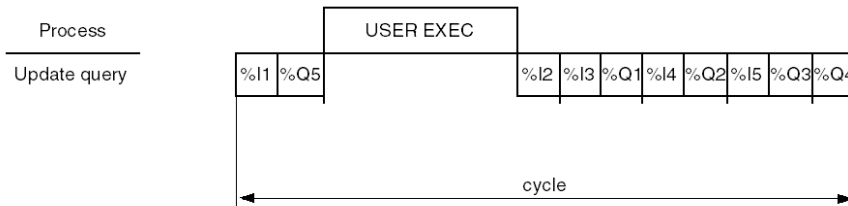
These stations are used with following RIO modules:

- 140 CRA 931 00
- 140 CRA 932 00

This system allows remote inputs/outputs to be updated on sections with optimum response times (without waiting for the entire task cycle before updating the inputs/outputs).

Operation

The following diagram shows the IO phases when 5 drops are associated to client task sections.



%I_i inputs of drop No. *i*

%Q_i outputs of drop No. *i*

i drop number

Description

| Phase | Description |
|-------|--|
| 1 | Request to update: <ul style="list-style-type: none"> • the inputs of the first drop (<i>i</i>=1) • the outputs of the last drop (<i>i</i>=5) |
| 2 | Processing the program |
| 3 | <ul style="list-style-type: none"> • Updating the inputs of the first drop (<i>i</i>=2) • Request to update the inputs of the second drop (<i>i</i>=2) |
| 4 | Request to update: <ul style="list-style-type: none"> • the inputs of the third drop (<i>i</i>=3) • the outputs of the first drop (<i>i</i>=1) |
| 5 | Request to update: <ul style="list-style-type: none"> • the inputs of the fourth drop (<i>i</i>=4) • the outputs of the second drop (<i>i</i>=2) |

| Phase | Description |
|-------|---|
| 6 | Request to update: <ul style="list-style-type: none"> the inputs of the last drop (i=5) the outputs of the third drop (i=3) |
| 7 | Request to update the outputs of the fourth drop (i=4) |

Adjustment of the Drop Hold-Up Time Value

In order for the remote outputs to be correctly updated and avoid fallback values to be applied, the drop hold-up time must be set to at least twice the mast task cycle time. Therefore the default value, 300 ms, must be changed if the MAST period is set to the maximum value, 255 ms. The adjustment of the Drop Hold-Up time (see Quantum using EcoStruxure™ Control Expert, Hot Standby System, User Manual) must be done on all configured drops.

Multitasking Execution

Subject of this Section

This section describes how a multitasking application operates.

Multitasking Software Structure

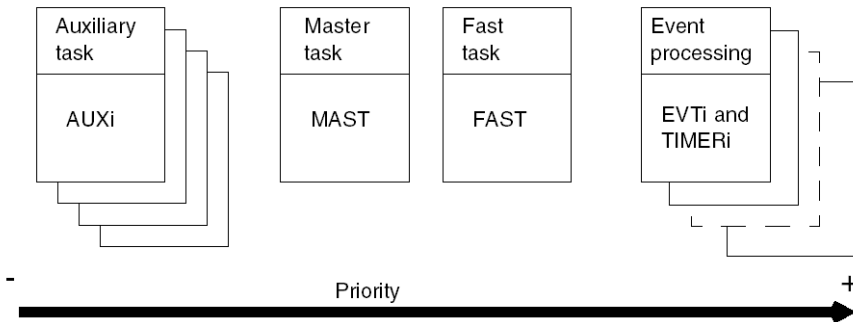
Tasks and Processing

The task structure of this type of application is as follows:

| Task/Processing | Designation | Description |
|-----------------|---------------------------------|---|
| Master | MAST | Always present, may be cyclic or periodic. |
| Fast | FAST | Optional, always periodic. |
| Auxiliary | AUX 0 to 3 | Optional and always periodic. |
| Event | EVTi and TIMERi, page 110 | Called by the system when an event occurs on an input/output module or triggered by the event timer. These types of processing are optional and can be used by applications that need to act on inputs/outputs within a short response time. |

Illustration

The following diagram shows the tasks in a multitasking structure and their level of priority.



Description

The master (MAST) task is still the application base. The other tasks differ depending on the type of PLC, page 80.

Levels of priority are fixed for each task in order to prioritize certain types of processing.

Event processing can be activated asynchronously with respect to periodic tasks by an order generated by external events. It is processed as a priority and requires any processing in progress to be stopped.

Subroutine limitations

Subroutines can only be used in one task. For example, **MAST** subroutines cannot be called from **TIMER** and **EVENT** tasks.

Precautions

⚠ CAUTION

UNEXPECTED MULTITASK APPLICATION BEHAVIOR

- The sharing of Inputs/Outputs between different tasks can lead to unforeseen behavior by the application.
- We specifically recommend you associate each output or each input to one task only.

Failure to follow these instructions can result in injury or equipment damage.

NOTE: During an update of %M linked to **FAST** task I/O, you must either:

- do them at the same time in the **FAST** task
- mask the **FAST** task (%S31) while updating

Sequencing of Tasks in a Multitasking Structure

General

The master task is active by default.

The fast and auxiliary tasks are active by default if they have been programmed.

Event processing is activated when the associated event occurs.

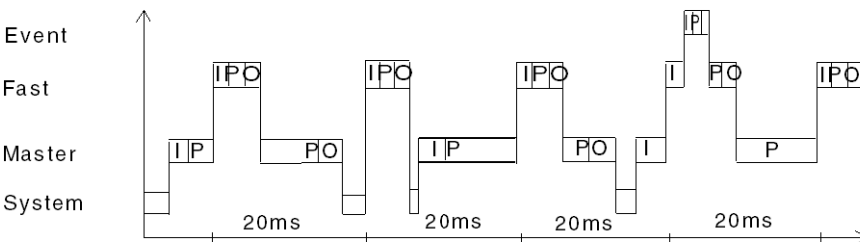
Operation

The table below describes the execution of priority tasks (this operation is also illustrated in the diagram below).

| Phase | Description |
|-------|---|
| 1 | Occurrence of an event or start of the fast task cycle. |
| 2 | Execution of lower priority tasks in progress stopped, |
| 3 | Execution of the priority task. |
| 4 | The interrupted task takes over again when processing of the priority task is complete. |

Description of the Task Sequence

The following diagram illustrates the task sequence of multitasking processing with a cyclic master task, a fast task with a 20ms period and event processing.



Legend:

I: acquisition of inputs

P: program processing

O: updating of outputs

Task Control

The execution of fast and event processing tasks can be controlled by the program using the following system bits:

- %S30 is used to control whether or not the MAST master task is active
- %S31 is used to control whether or not the FAST task is active..
- %S32 to %S35 are used to control whether or not the auxiliary tasks AUX0 to AUX3 are active.
- %S38 is used to control whether EVTi event processing is active.

NOTE: The elementary functions MASKEVT and UNMASKEVT also allow the global masking and unmasking of events by the program.

Task Control

Cyclic and Periodic Operation

In multitasking operation, the highest priority task shall be used in periodic mode in order to allow enough time for lower priority tasks to be executed.

For this reason, only the task with the lowest priority should be used in cyclic mode. Thus, choosing cyclic operating mode for the master task excludes using auxiliary tasks.

Measurement of Task Durations

The duration of tasks is continually measured. This measurement represents the duration between the start and the end of execution of the task. This measurement includes the time taken up by tasks of higher priority which may interrupt the execution of the task being measured.

The following system words (see EcoStruxure™ Control Expert, System Bits and Words, Reference Manual) give the current, maximum and minimum cycle times for each task (value in ms)

| Measurement of times | MAST | FAST | AUX0 | AUX1 | AUX2 | AUX3 |
|----------------------|-------|-------|-------|-------|-------|-------|
| Current | %SW30 | %SW33 | %SW36 | %SW39 | %SW42 | %SW45 |
| Maximum | %SW31 | %SW34 | %SW37 | %SW40 | %SW43 | %SW46 |
| Minimum | %SW32 | %SW35 | %SW38 | %SW41 | %SW44 | %SW47 |

NOTE: The maximum and minimum times are taken from the times measured since the last cold restart.

Task Periods

The task periods are defined in the task properties. They can be modified by the following system words.

| System words | Task | Values | Default values | Observations |
|--------------|------|-------------|----------------|---|
| %SW0 | MAST | 0..255ms | Cyclic | 0 = cyclic operation |
| %SW1 | FAST | 1..255ms | 5ms | - |
| %SW2 | AUX0 | 10ms..2.55s | 100ms | The values of the period are expressed in 10ms. |
| %SW3 | AUX1 | 10ms..2.55s | 200ms | |
| %SW4 | AUX2 | 10ms..2.55s | 300ms | |
| %SW5 | AUX3 | 10ms..2.55s | 400ms | |

When the cycle time of the task exceeds the period, the system sets the system bit %S19 of the task to 1 and continues with the following cycle.

NOTE: The values of the periods do not depend on the priority of tasks. It is possible to define the period of a fast task which is larger than the master task.

Watchdog

The execution of each task is controlled by a configurable watchdog by using the task properties.

The following table gives the range of watchdog values for each of the tasks:

| Tasks | Watchdog values (min... max) (ms) | Default watchdog value (ms) | Associated system word |
|-------|-----------------------------------|-----------------------------|------------------------|
| MAST | 10..1500 | 250 | %SW11 |
| FAST | 10..500 | 100 | - |
| AUX0 | 100..5000 | 2000 | - |

| Tasks | Watchdog values (min... max) (ms) | Default watchdog value (ms) | Associated system word |
|-------|-----------------------------------|-----------------------------|------------------------|
| AUX1 | 100..5000 | 2000 | - |
| AUX2 | 100..5000 | 2000 | - |
| AUX3 | 100..5000 | 2000 | - |

If watchdog overflow should occur, the application is declared in error, which causes the PLC to stop immediately (HALT state).

The word %SW11 contains the watchdog value of the master task in ms. This value is not modifiable by the program.

The bit %S11 indicates a watchdog overflow. It is set to 1 by the system when the cycle time becomes greater than the watchdog.

NOTE:

- The reactivation of the task requires the terminal to be connected in order to analyze the cause of the error, correct it, reinitialize the PLC and switch it to RUN.
- It is not possible to exit HALT by switching to STOP. To do this you must reinitialize the application to ensure consistency of data.

Task Control

When the application program is being executed, it is possible to activate or inhibit a task by using the following system bits:

| System bits | Task |
|-------------|------|
| %S30 | MAST |
| %S31 | FAST |
| %S32 | AUX0 |
| %S33 | AUX1 |
| %S34 | AUX2 |
| %S35 | AUX3 |

The task is active when the associated system bit is set to 1. These bits are tested by the system at the end of the master task.

When a task is inhibited, the inputs continue to be read and the outputs continue to be written.

On startup of the application program, for the first execution cycle only the master task is active. At the end of the first cycle the other tasks are automatically activated except if one of the tasks is inhibited (associated system bit set to 0) by the program.

Controls on Input Reading and Output Writing Phases

The bits of the following system words can be used (only when the PLC is in RUN) to inhibit the input reading and output writing phases.

| Inhibition of phases... | MAST | FAST | AUX0 | AUX1 | AUX2 | AUX3 |
|-------------------------|----------|----------|----------|----------|----------|----------|
| reading of inputs | %SW8 . 0 | %SW8 . 1 | %SW8 . 2 | %SW8 . 3 | %SW8 . 4 | %SW8 . 5 |
| writing of outputs | %SW9 . 0 | %SW9 . 1 | %SW9 . 2 | %SW9 . 3 | %SW9 . 4 | %SW9 . 5 |

NOTE: By default, the input reading and output writing phases are active (bits of system words %SW8 and %SW9 set to 0).

On Quantum, inputs/outputs which are distributed via DIO bus are not assigned by the words %SW8 and %SW9.

Assignment of Input/Output Channels to Master, Fast and Auxiliary Tasks

General

Each task writes and reads the inputs/outputs assigned to it.

The association of a channel, group of channels or an input/output module with a task is defined in the configuration screen of the corresponding module.

The task that is associated by default is the MAST task.

Reading of Inputs and Writing of Outputs on Premium

All the input/output channels of in-rack modules can be associated with a task (MAST, FAST or AUX 0..3).

Local and remote inputs/outputs (X bus):

For each task cycle, the inputs are read at the start of the task and the outputs are written at the end of the task.

Remote inputs/outputs on Fipio bus:

In controlled mode, the refreshing of inputs/outputs is correlated with the task period. The system guarantees that inputs/outputs are updated in a single period. Only the inputs/outputs associated with this task are refreshed.

In this mode, the period of the PLC task (MAST, FAST or AUX) must be greater than or equal to the network cycle time.

In free mode, no restriction is imposed on the task period. The PLC task period (MAST, FAST or AUX) can be less than the network cycle. If this is the case, the task can be executed without updating the inputs/outputs. Selecting this mode gives you the possibility of having the lowest possible task times for applications where speed is critical.

Example on Premium

With its 8 successive channel modularity (channels 0 to 7, channels 8 to 15, etc.), the inputs/outputs of the Premium discrete modules can be assigned in groups of 8 channels, independently of the MAST, AUXi or FAST task.

Example: it is possible to assign the channels of a 28 input/output module as follows:

- inputs 0 to 7 assigned to the MAST task,
- inputs 8 to 15 assigned to the FAST task,
- outputs 0 to 7 assigned to the MAST task,
- outputs 8 to 15 assigned to the AUX0 task.

Reading of Inputs and Writing of Outputs on Quantum

Local inputs/outputs:

Each input/output module or group of modules can be associated with a single task (MAST, FAST or AUX 0..3).

Remote inputs/outputs:

Remote input/output stations can only be associated with the master (MAST) task. The assignment is made for sections, page 100, with 1 remote input station and 1 remote output station per section.

Distributed inputs/outputs:

Distributed input/output stations can only be associated with the master (MAST) task.

The inputs are read at the start of the master task and the outputs are written at the end of the master task.

Reading of Inputs and Writing of Outputs on M580

Local inputs/outputs:

Each input/output module or group of modules can be associated with a single task (MAST, FAST, AUX0 or AUX1).

Remote Inputs/Outputs:

The tasks available to be associated to remote inputs and outputs depend upon the adapter module installed in the remote rack (see Modicon M580, RIO Modules, Installation and Configuration Guide).

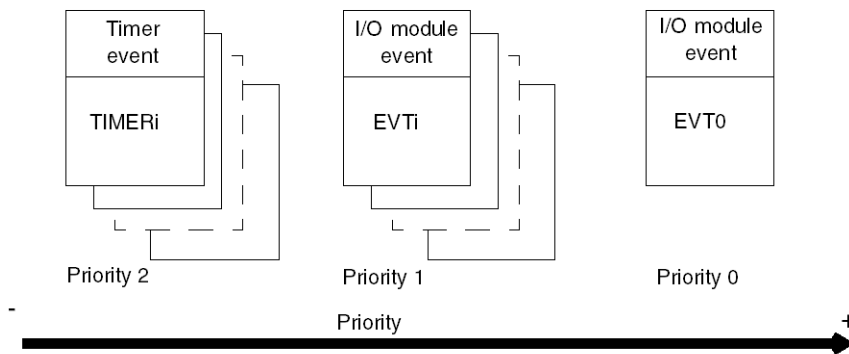
Distributed Inputs/Outputs: Distributed inputs and outputs can be associated only with the master (MAST) task.

Management of Event Processing

General

Event processing take priority over tasks.

The following illustration describes the 3 defined levels of priority:



Management of Priorities

- EVT0 event processing is the highest priority processing. It can itself interrupt other types of event processing.
- EVTi event processing triggered by input/output modules (priority 1) take priority over TIMERi event processing triggered by timers (priority 2).
- **On Modicon M580, M340, Premium and Atrium PLCs:** types of event processing with priority level 1 are stored and processed in order.
- **On Quantum PLC:** the priority of priority 1 processing types is determined:
 - by the position of the input/output module in the rack,
 - by the position of the channel in the module.

The module with the lowest position number has the highest level of priority.

- Event processing triggered by timer is given priority level 2. The processing priority is determined by the lowest timer number.

Control

The application program can globally validate or inhibit the various types of event processing by using the system bit %S38. If one or more events occur while they are inhibited, the associated processing is lost.

Two elementary functions of the language, `MASKEVT ()` and `UNMASKEVT ()`, used in the application program can also be used to mask or unmask event processing.

If one or more events occur while they are masked, they are stored by the system and the associated processing is carried out after unmasking.

Execution of TIMER-type Event Processing

Description

TIMER-type event processing is any process triggered by the `ITCNTRL` (see EcoStruxure™ Control Expert, System, Block Library) function.

This timer function periodically activates event processing every time the preset value is reached.

Reference

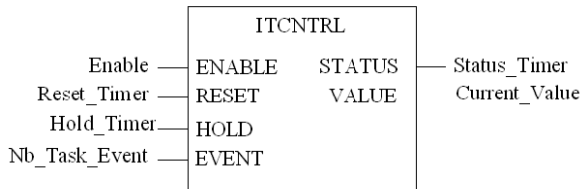
The following parameters are selected in the event processing properties.

| Parameter | Value | Default value | Role |
|-----------|--------------------------|---------------|--|
| Time base | 1 ms, 10ms, 100ms, 1 sec | 10ms | Timer time base. Note: the time base of 1ms should be used with care, as there is a risk of overrun if the processing triggering frequency is too high. |
| Preset | 1..1023 | 10 | Timer preset value. The time period obtained equals: Preset x Time Base. |
| Phase | 0..1023 | 0 | The value of the temporal offset between the STOP/RUN transition of the PLC and the first restart of the timer from 0. The temporal value equals: Phase x Time Base. |

NOTE: The Phase must be lower than Preset in TIMER-type Event.

ITCNTRL Function

Representation in FBD:



The following table describes the input parameters:

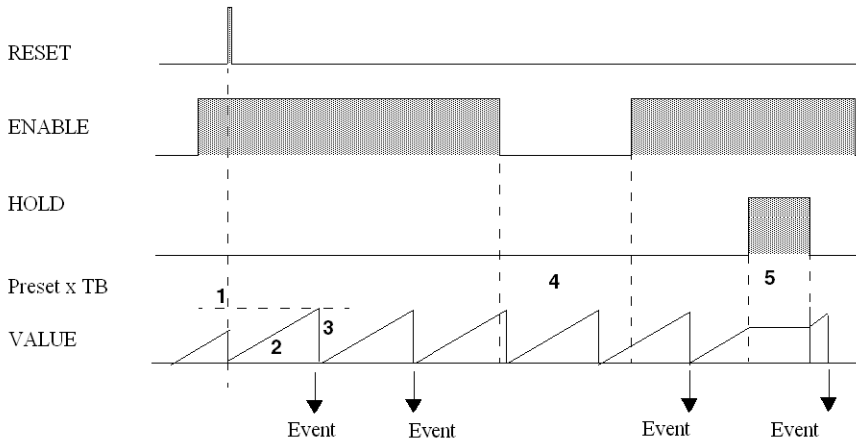
| Parameter | Type | Comment |
|---------------|------|--|
| Enable | BOOL | Enable input selected |
| Reset_Timer | BOOL | At 1 resets the timer |
| Hold_Timer | BOOL | At 1, freezes timer incrementation. |
| Nb_Task_Event | BYTE | Input byte which determines the event processing number to be triggered. |

The following table describes the output parameters:

| Parameter | Type | Comment |
|---------------|------|-------------------------|
| Status_Timer | WORD | Status word. |
| Current_Value | TIME | Current value of timer. |

Timing Diagram for Normal Operation

Timing diagram.



Normal operation

The following table describes the triggering of TIMER-type event processing operations (see timing diagram above).

| Phase | Description |
|-------|---|
| 1 | When a rising edge is received on the <code>RESET</code> input, the timer is reset to 0. |
| 2 | The current value <code>VALUE</code> of the timer increases from 0 towards the preset value at a rate of one unit for each pulse of the time base. |
| 3 | An event is generated when the current value has reached the preset value, the timer is reset to 0, and then reactivated. The associated event processing is also triggered, if the event is not masked. It can be deferred if an event processing task with a higher or identical priority is already in progress. |
| 4 | When the <code>ENABLE</code> input is at 0, the events are no longer sent out. TIMER type event processing is no longer triggered. |
| 5 | When the <code>HOLD</code> input is at 1, the timer is frozen, and the current value stops incrementing, until this input returns to 0. |

Event Processing Synchronization

The Phase parameter is used to trigger different TIMER-type event processing tasks at constant time intervals.

This parameter set a temporal offset value with an absolute time origin, which is the last passage of the PLC from STOP to RUN.

Operating condition:

- The event processing tasks must have the same time base and preset values.
- The RESET and HOLD inputs must not be set to 1.

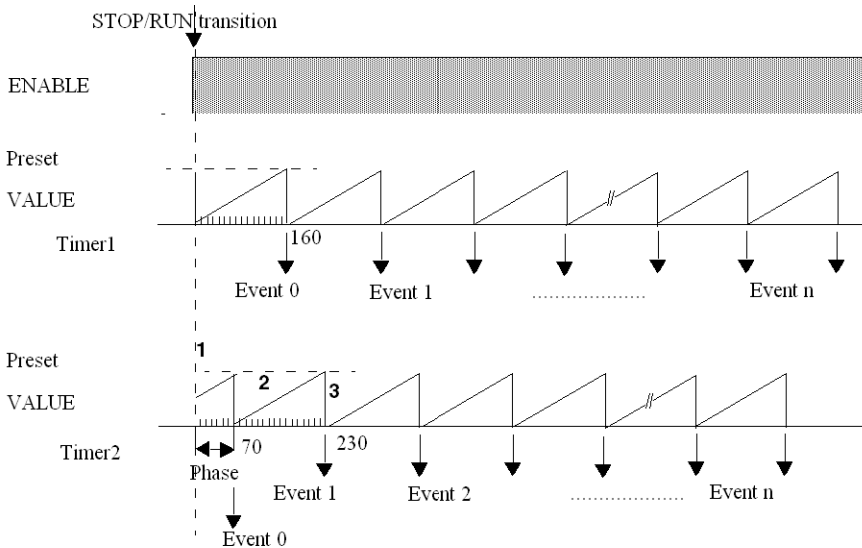
Example: Two event processing tasks Timer1 and Timer2 to be executed at 70ms interval.

Timer1 can be defined with a phase equal to 0 and the second Timer2 with a phase of 70ms (phase of 7 and time base of 10ms).

Any event triggered by the timer associated with the Timer1 processing task shall be followed after an interval of 70ms by an event from the timer associated with the Timer2 processing task

Timing Diagram: STOP/RUN Transition

Timing diagram of the example provided above with the same preset value of 16 (160ms) for Timer1 and Timer2.



Operation After PLC STOP/RUN

The following table describes the operation of the PLC after a transition from STOP into RUN (see timing diagram above):

| Phase | Description |
|-------|--|
| 1 | ON a STOP RUN transition of the PLC, timing is triggered so that the preset value is reached at the end of a time period equal to Phase x time base, when the first event is sent out. |
| 2 | The current value <code>VALUE</code> of the timer increases from 0 towards the preset value at a rate of one unit for each pulse of the time base. |
| 3 | An event is generated when the current value has reached the preset value, the timer is reset to 0, and then reactivated. The associated event processing is also triggered, if the event is not masked. If can be deferred, if there is an event processing task of higher or identical priority already in progress. |

Input/Output Exchanges in Event Processing

General

With each type of event processing it is possible to use other input/output channels than those for the event.

As with tasks, exchanges are then performed implicitly by the system before (%I) and after (%Q) application processing.

Operation

The following table describes the exchanges and processing performed.

| Phase | Description |
|-------|--|
| 1 | The occurrence of an event reroutes the application program to perform the processing associated with the input/output channel which caused the event. |
| 2 | All inputs associated with event processing are acquired automatically. |
| 3 | The event processing is executed. It must be as short as possible. |
| 4 | All the outputs associated with the event processing are updated. |

Premium/Atrium PLCs

The inputs acquired and the outputs updated are:

- the inputs associated with the channel which caused the event
- the inputs and outputs used during event processing

NOTE: These exchanges may relate:

- to a channel (e.g. counting module) or
- to a group of channels (discrete module). In this case, if the processing modifies, for example, outputs 2 and 3 of a discrete module, the image of outputs 0 to 7 is then transferred to the module.

Quantum PLCs

The inputs acquired and the outputs updated are selected in the configuration. Only local inputs/outputs can be selected.

Programming Rule

The inputs (and the associated group of channels) exchanged during the execution of event processing are updated (loss of historical values, and thus edges). You should therefore avoid testing fronts on these inputs in the master (MAST), fast (FAST) or auxiliary (AUXi) tasks.

How to Program Event Processing

Procedure

The table below summarizes the essential steps for programming event processing.

| Step | Action |
|------|---|
| 1 | <p>Configuration phase (for events triggered by input/output modules)</p> <p>In offline mode, from the configuration editor, select Event Processing (EVT) and the event processing number for the channel of the input/output module concerned.</p> |
| 2 | <p>Unmasking phase</p> <p>The task which can be interrupted must in particular:</p> <ul style="list-style-type: none">• Enable processing of events at system level: set bit %S38 to 1 (default value).• Unmask events with the instruction UNMASKEVT (active by default).• Unmask the events concerned at channel level (for events triggered by input/output modules) by setting the input/output module's implicit language objects for unmasking of events to 1. By default, the events are masked.• Check that the stack of events at system level is not saturated (bit %S39 must be at 0). |
| 3 | <p>Event program creation phase</p> <p>The program must:</p> <ul style="list-style-type: none">• Determine the origin of the event(s) on the basis of the event status word associated with the input/output module if the module is able to generate several events.• Carry out the reflex processing associated with the event. This process must be as short as possible.• Write the reflex outputs concerned. <p>Note: the event status word is automatically reset to zero.</p> |

Illustration of Event Unmasking

This figure shows event unmasking in the MAST task.

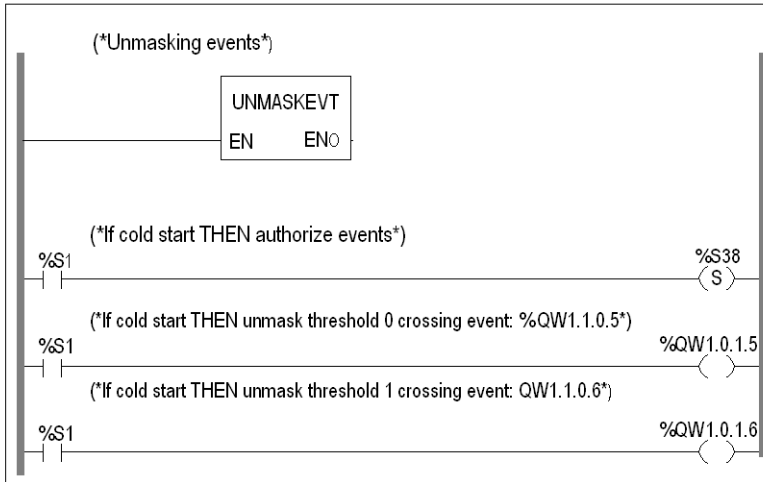
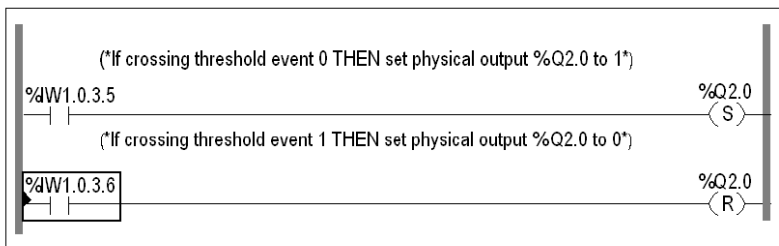


Illustration of the Contents of Event Processing

This figure shows the possible contents of event processing (bit test and action).



Application Memory Structure

What's in This Chapter

| | |
|--|-----|
| Input Output Data Addressing Methods | 119 |
| Memory Structure of the Premium, Atrium and Modicon M340 | |
| PLCs | 123 |
| Memory Structure of Quantum PLCs | 130 |

Subject of this Chapter

This chapter describes the CPU application memory structure.

Input Output Data Addressing Methods

Subject of This Section

This section presents input/output data addressing methods allowed by platform.

Input Output Data Addressing Methods

Introduction

The addressing method of data associated with controller input/output depends on the platform, I/O location, and topology.

A summary of addressing methods is provided for the following platforms:

- Modicon M580, page 119
- Modicon M340, page 120
- Modicon Quantum, page 121
- Modicon Premium, page 121
- Modicon Momentum, page 122

Modicon M580

For information about the Modicon M580 application memory structure, refer to the chapter on BME P58 xxxx CPU Memory Structure (see Modicon M580, Hardware, Reference Manual).

Data addressing method allowed depending on the module location in the architecture:

| Addressing (addressing example) | | Local rack | RIO Drop | | | DIO | CANopen | ASI | Profibus |
|--|-----------------------|------------|----------|-------------|-------|-----|---------|-----|----------|
| | | | EIO x80 | EIO Quantum | S9-08 | | | | |
| Located I/O | | | | | | | | | |
| Topological Addressing | Topological (%lr.m.c) | X | - | - | X | - | - | X | - |
| | IODDT (%CHr.m.c) | X | - | - | X | - | - | X | - |
| Flat or Modbus Addressing | State RAM (%lx) | - | - | X | X | - | - | - | - |
| | Located Memory (%MWx) | - | - | X | X | - | - | - | - |
| Unlocated I/O | | | | | | | | | |
| Device DDT PLC0_dx_ry_sz_Module, page 205 | | X | X | X | - | X | X | X | X |
| <p>X Allowed addressing method.</p> <p>- Not allowed addressing method.</p> | | | | | | | | | |

Modicon M340

Data addressing method allowed depending on the module location in the architecture:

| Addressing (addressing example) | | Local rack | DIO (NOE scan) | DIO (NOC scan) | CANopen | ASI | Profibus |
|---------------------------------|-----------------------|------------|----------------|----------------|---------|-----|----------|
| Located I/O | | | | | | | |
| Topological Addressing | Topological (%lr.m.c) | X | - | - | X | X | - |
| | IODDT (%CHr.m.c) | X | - | - | X | X | - |
| Flat or Modbus Addressing | State RAM (%lx) | X | - | - | - | - | - |

| | | | | | | | |
|---|--|---|---|---|---|---|---|
| | Located Memory (%MWx) | X | X | X | X | – | X |
| Unlocated I/O | | | | | | | |
| | Device DDT PLC0_dx_ry_sz_Module, page 205 | – | – | – | – | – | – |
| X Allowed addressing method. – Not allowed addressing method. | | | | | | | |

Modicon Quantum

Data addressing method allowed depending on the module location in the architecture:

| Addressing (addressing example) | | Local rack | RIO Drop | | | DIO (NOE scan) | DIO (NOC scan) |
|---|--|------------|----------|-------------|------|----------------|----------------|
| | | | EIO x80 | EIO Quantum | S908 | | |
| Unlocated I/O | | | | | | | |
| Topological Addressing | Topological (%lr.m.c) | X | – | – | X | – | – |
| | IODDT (%CHr.m.c) | X | – | – | X | – | – |
| Flat or Modbus Addressing | State RAM (%lx) | X | – | X | X | X | – |
| | Located Memory (%MWx) | X | – | X | X | X | X |
| Unlocated I/O | | | | | | | |
| | Device DDT PLC0_dx_ry_sz_Module, page 205 | – | X | – | – | – | – |
| X Allowed addressing method. – Not allowed addressing method. | | | | | | | |

Modicon Premium

Data addressing method allowed depending on the module location in the architecture:

| Addressing (addressing example) | | Local rack | DIO (ETY scan) | DIO (ETC scan) | CANopen |
|--|-----------------------|------------|----------------|----------------|---------|
| Located I/O | | | | | |
| Topological Addressing | Topological (%lr.m.c) | X | - | - | - |
| | IODDT (%CHR.m.c) | X | - | - | - |
| Flat or Modbus Addressing | State RAM (%lx) | - | - | - | - |
| | Located Memory (%MWx) | - | X | X | X |
| Unlocated I/O | | | | | |
| Device DDT PLC0_dx_ry_sz_Module, page 205 | | - | - | - | - |
| <p>X Allowed addressing method.</p> <p>- Not allowed addressing method.</p> | | | | | |

Modicon Momentum

Data addressing method allowed depending on the module location in the architecture:

| Addressing (addressing example) | | Momentum Bus + I/O-Bus |
|---------------------------------|-----------------------|------------------------|
| Located I/O | | |
| Topological Addressing | Topological (%lr.m.c) | - |
| | IODDT (%CHR.m.c) | - |
| Flat or Modbus Addressing | State RAM (%lx) | X |
| | Located Memory (%MWx) | X |

| | |
|---|---|
| Unlocated I/O | |
| Device DDT PLC0_dx_ry_sz_Module, page 205 | – |
| X Allowed addressing method. – Not allowed addressing method. | |

Memory Structure of the Premium, Atrium and Modicon M340 PLCs

Subject of this Section

This section describes memory structure and detailed description of the memory zones of the Modicon Premium, Atrium and M340 PLCs.

Memory Structure of Modicon M340 PLCs

Overview

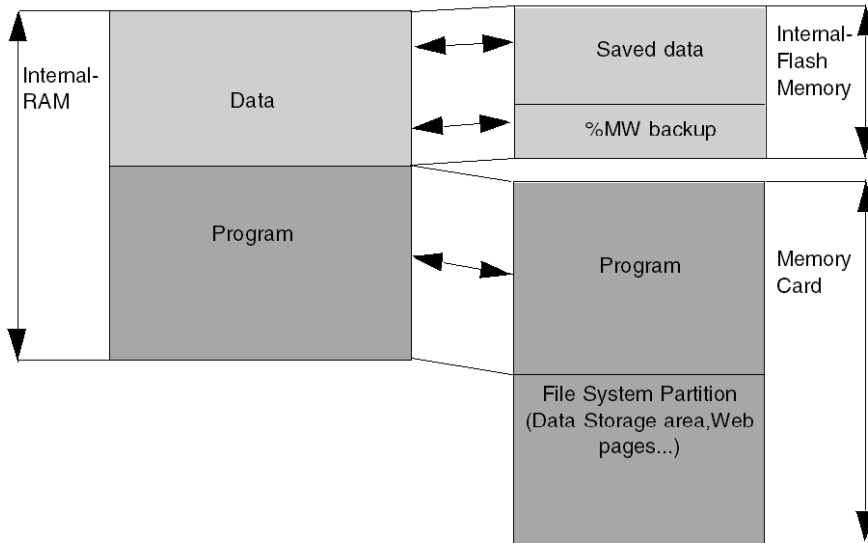
The PLC memory supports:

- **located application data**
- **unlocated application data**
- **the program**: task descriptors and executable code, constant words, initial values and configuration of inputs/outputs

Structure

The data and program are supported by the processor module's internal RAM.

The following diagram describes the memory structure.



State RAM

For **Data** (see graphic above) also State RAM is available, if you select **Mixed topological and State RAM** in the **Configuration** tab of a *Modicon M340 processor* (see *EcoStruxure™ Control Expert, Operating Modes*).

To use this option you need Modicon M340 firmware 2.4 or later.

NOTE:

If you want to import a legacy LL984 Compact application which uses Modbus request to communicate with an HMI, you have to use State RAM addressing to preserve the Modbus exchange between PLC and HMI.

The State RAM contains the following located data:

| Address | Object address | Data use |
|---------|-------------------|---|
| 0xxxxx | %Qr.m.c.d,%Mi | output module bits and internal bits |
| 1xxxxx | %Ir.m.c.d, %Ii | input module bits |
| 3xxxxx | %IW r.m.c.d, %IWi | input words of input/output modules |
| 4xxxxx | %QW r.m.c.d, %MWi | output words of input/output modules and internal words |

NOTE: Not all data represented in topological addressing is available in State RAM.

Please refer to *Topological/State RAM Addressing of Modicon M340 Discrete Modules* (see Modicon X80, Discrete Input/Output Modules, User Manual) and *Topological/State RAM Addressing of Modicon M340 Analog Modules* (see Modicon X80, Analog Input/Output Modules, User Manual).

Program Backup

If the memory card is present, working properly and not write-protected, the program is saved on the memory card:

- Automatically, after:
 - a download
 - online modification
 - a rising edge of the system bit %S66 in the project program
- Manually:
 - with the command **PLC > Project backup > Backup Save**
 - in an animation table by setting the system bit %S66

WARNING

LOSS OF DATA - APPLICATION NOT SAVED

The interruption of an application saving procedure by an untimely or rough extraction of the memory card, may lead to the loss of saved application. The bit %S65 allows managing a correct extraction.

Failure to follow these instructions can result in death, serious injury, or equipment damage.

NOTE: For detail on %S65, refer to chapter *System Bits* (see EcoStruxure™ Control Expert, System Bits and Words, Reference Manual).

The memory card uses Flash technology, therefore no battery is necessary.

Program Restore

If the memory card is present and working properly, the program is copied from the PLC memory card to the internal memory:

- Automatically after:
 - a power cycle
- Manually, with the Control Expert command **PLC > Project backup > Backup Restore**

NOTE: When you insert the memory card in run or stop mode, you have to do a power cycle to restore the project on the PLC.

Saved Data

Located, unlocated data, diagnostic buffer are automatically saved in the internal Flash memory at power-off. They are restored at warm start.

Save_Param

The `SAVE_PARAM` function does both current and initial parameter adjustment in internal RAM (as in other PLCs). In this case, the internal RAM and the memory card content are different (`%S96 = 0` and the `CARDERR` LED is on). On cold start (after application restore), the current parameter are replaced by the last adjusted initial values only if a save to memory card function (Backup Save or `%S66` rising edge) was done.

Save Current Value

On a `%S94` rising edge, the current values replace the initial values in internal memory. The internal RAM and the memory card content are different (`%S96 = 0` and the `CARDERR` LED is on). On cold start, the current values are replaced by the most recent initial values only if a save to memory card function (Backup Save or `%S66` rising edge) was done.

Delete Files

There are two ways to delete all the files on the memory card:

- Erasing the memory card (delete all files of the file system partition)
- Deleting the content of directory `\DataStorage\` (delete only files added by user)

Both actions are performed using `%SW93` (see `EcoStruxure™ Control Expert, System Bits and Words, Reference Manual`).

The system word `%SW93` can only be used after download of a default application in the PLC.

CAUTION

INOPERABLE MEMORY CARD

Do not format the memory card with a non-Schneider tool. The memory card needs a structure to contain program and data. Formatting with another tool destroys this structure.

Failure to follow these instructions can result in injury or equipment damage.

%MW Backup

The values of the %MW_i can be saved in the internal Flash memory using %SW96 (see EcoStruxure™ Control Expert, System Bits and Words, Reference Manual). These values will be restored at cold start, including application download, if the option **Initialize of %MW on cold start** is unchecked in the processor Configuration screen (see EcoStruxure™ Control Expert, Operating Modes).

For %MW words, the values can be saved and restored on cold restart or download if the option **Reset of %MW on cold restart** is not checked in the processor Configuration screen. With the %SW96 word, management of memory action %MW internal words (save, delete) and information on the actions' states %MW internal words is possible.

Memory Card Specifics

Two types of memory card are available:

- **application**: these cards contain the application program and Web pages
- **application + file storage**: these cards contain the application program, data files from Memory Card File Management EFBs, and Web pages

Memory Structure of Premium and Atrium PLCs

General

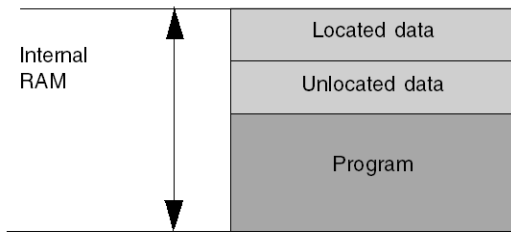
The PLC memory supports:

- **located application data**,
- **unlocated application data**,
- **the program**: task descriptors and executable code, constant words, initial values and configuration of inputs/outputs.

Structure without Memory Extension Card

The data and program are supported by the internal RAM of the processor module.

The following diagram describes the memory structure.

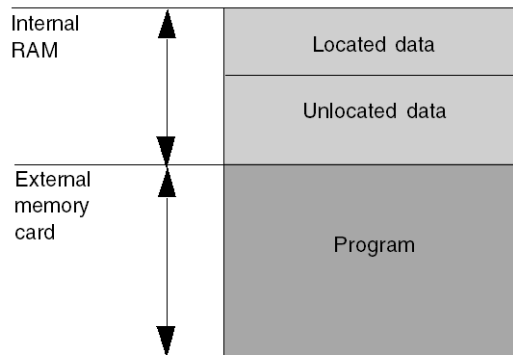


Structure with Memory Extension Card

The data is supported by the internal RAM of the processor module.

The program is supported by the extension memory card.

The following diagram describes the memory structure.



Memory Backup

The internal RAM is backed up by a Ni-Cad battery supported by the processor module.

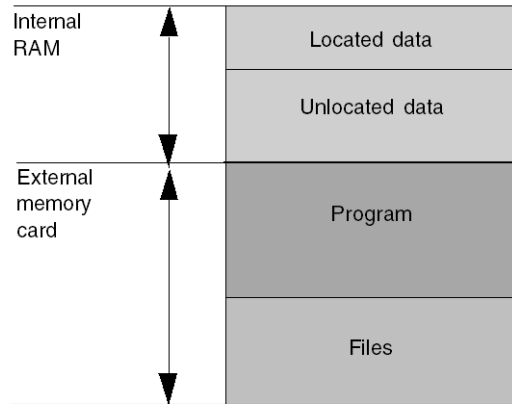
The RAM memory cards are backed up by a Ni-Cad battery.

Specificities of Memory Cards

Three types of memory card are offered:

- **application:** these cards contain the application program. The cards offered use either RAM or Flash EPROM technology
- **application + file storage:** in addition to the program, these cards also contain a zone which can be used to backup/restore data using the program. The cards on offer use either RAM or Flash EPROM technology
- **file storage:** these cards can be used to backup/restore data using the program. These cards use SRAM technology.

The following diagram describes the memory structure with an application and file storage card.



NOTE: On processors with 2 memory card slots, the lower slot is reserved for the file storage function.

Detailed Description of the Memory Zones

User Data

This zone contains the located and unlocated application data.

- located data:
 - %M, %S Boolean and %MW,%SW numerical data
 - data associated with modules (%I, %Q, %IW, %QW,%KW etc.)
- unlocated data:
 - Boolean and numerical data (instances)
 - EFB and DFB instances

User Program and Constants

This zone contains the executable codes and constants of the application.

- executable codes:
 - program code
 - code associated with EFs, EFBs and the management of I/O modules
 - code associated with DFBs
- constants:
 - KW constant words
 - constants associated with inputs/outputs
 - initial data values

This zone also contains the necessary information for downloading the application: graphic codes, symbols etc.

Other Information

Other information relating to the configuration and structure of the application are also stored in the memory (in a data or program zone depending on the type of information).

- Configuration: other data relating to the configuration (hardware configuration, software configuration).
- System: data used by the operating system (task stack, etc.).
- Diagnostics: information relating to process or system diagnostics, diagnostics buffer.

Memory Structure of Quantum PLCs

Subject of this Section

This section describes memory structure and detailed description of the memory zones of the Quantum PLCs.

Memory Structure of Quantum PLCs

General

The PLC memory supports:

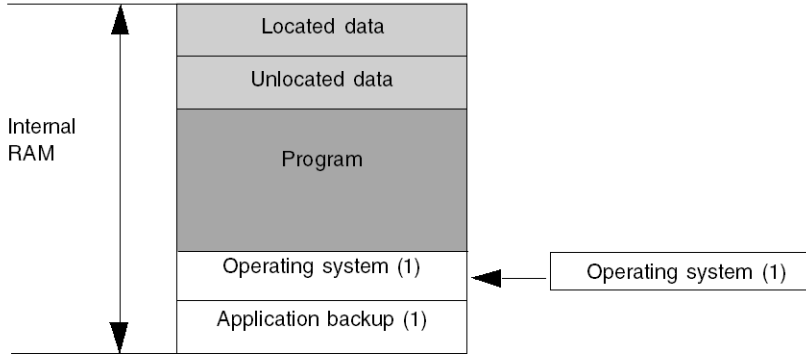
- **located application data** (State Ram),
- **unlocated application data**,

- **the program:** task descriptors and executable code, initial values and configuration of inputs/outputs.

Structure without Memory Extension Card

The data and program are supported by the internal RAM of the processor module.

The following diagram describes the memory structure.



(1) Only for 140 CPU 31••/43••/53•• processors.

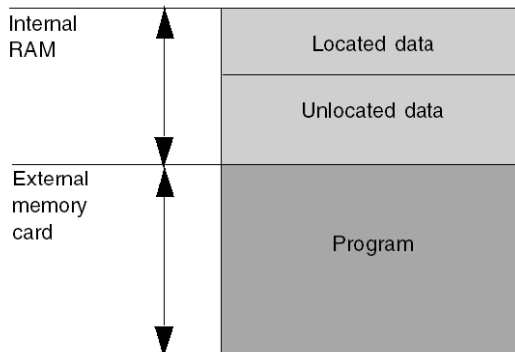
Structure with Memory Extension Card

Quantum 140 CPU 6••• processors can be fitted with a memory extension card.

The data is supported by the internal RAM of the processor module.

The program is supported by the extension memory card.

The following diagram describes the memory structure.



Memory Backup

The internal RAM is backed up by a Ni-Cad battery supported by the processor module.

The RAM memory cards are backed up by a Ni-Cad battery.

Start-up with Application Saved in Backup Memory

The following table describes the different results according to the PLC state, according to the PLC mem switch (see Quantum using EcoStruxure™ Control Expert, Hardware, Reference Manual), and also indicates if the box "Auto RUN" is checked or not checked.

| PLC State | PLC Mem Switch ¹ | Auto RUN in Appl ² | Results |
|------------|-----------------------------|-------------------------------|--|
| NONCONF | Start or Off | Off | Cold Start , application is loaded from Backup memory to RAM of the PLC. The PLC remains in STOP. |
| NONCONF | Start or Off | On | Cold Start , application is loaded from Backup memory to RAM of the PLC. The PLC remains in RUN. |
| NONCONF | Mem Prt or Stop | Not Applicable | No application loaded. PLC power up in NONCONF state. |
| Configured | Start or Off | Off | Cold Start , application is loaded from Backup memory to RAM of the PLC. The PLC remains in STOP. |
| Configured | Start or Off | On | Cold Start , application is loaded from Backup memory to RAM of the PLC. The PLC remains in RUN. |
| Configured | Mem Prt or Stop | Do not Care | Warm Start , no application loaded. PLC powers up in previous state. |

¹ Start and Stop are valid for the 434 and 534 models only and Off is valid for the 311 only. Mem Prt is valid on all models.

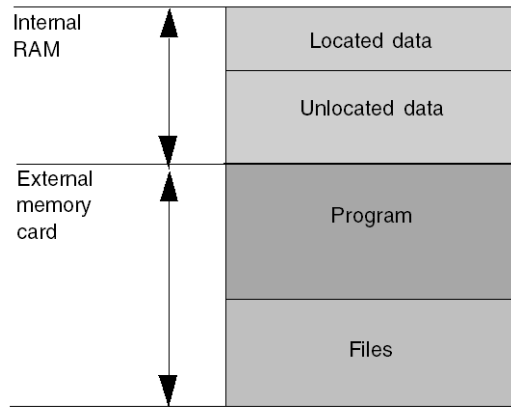
² The Automatic RUN in the application refers to the application that is loaded.

Specificities of Memory Cards

Three types of memory card are offered:

- **application**: these cards contain the application program. The cards on offer use either RAM or Flash EPROM technology
- **application + file storage**: in addition to the program, these cards also contain a zone which can be used to backup/restore data using the program. The cards on offer use either RAM or Flash EPROM technology
- **file storage**: these cards can be used to backup/restore data using the program. These cards use SRAM technology.

The following diagram describes the memory structure with an application and file storage card.



NOTE: On processors with 2 memory card slots, the lower slot is reserved for the file storage function.

Detailed Description of the Memory Zones

Unlocated Data

This zone contains unlocated data:

- Boolean and numerical data
- EFB and DFB

Located Data

This zone contains located data (State Ram):

| Address | Object address | Data use |
|---------|------------------|--|
| 0xxxxx | %Qr.m.c.d,%Mi | output module bits and internal bits. |
| 1xxxxx | %lr.m.c.d, %li | input module bits. |
| 3xxxxx | %lWr.m.c.d, %lWi | input register words of input/output modules. |
| 4xxxxx | %QWr.m.c.d, %MWi | output words of input/output modules and internal words. |

User Program

This zone contains the executable codes of the application.

- program code
- code associated with EFs, EFBs and the management of I/O modules
- code associated with DFBs
- initial variable values

This zone also contains the necessary information for downloading the application: graphic codes, symbols etc.

Operating System

On 140 CPU 31••/41••/51•• processors, this contains the operating system for processing the application. This operating system is transferred from an internal EPROM memory to internal RAM on power up.

Application Backup

A Flash EPROM memory zone of 1435K8, available on processors 140 CPU 31••/41••/51••, can be used to backup the program and the initial values of variables.

The application stored in this zone is automatically transferred to internal RAM when the PLC processor is powered up (if the PLC MEM switch is set to off on the processor front panel).

Other Information

Other information relating to the configuration and structure of the application are also stored in the memory (in a data or program zone depending on the type of information).

- Configuration: other data relating to the configuration (hardware configuration, software configuration).
- System: data used by the operating system (task stack, etc.).
- Diagnostics: information relating to process or system diagnostics, diagnostics buffer.

Operating Modes

What's in This Chapter

| | |
|---|-----|
| Modicon M340 PLCs Operating Modes..... | 135 |
| Premium, Quantum PLCs Operating Modes | 145 |
| PLC HALT Mode | 156 |

Subject of this Chapter

The chapter describes the operating modes of the PLC in the event of power outage and restoral, the impacts on the application program and the updating of inputs/outputs.

For information about the Modicon M580, refer to BME P58 CPUs Operating Modes (see Modicon M580, Hardware, Reference Manual).

Modicon M340 PLCs Operating Modes

Subject of this Section

This section describes the operating modes of the Modicon M340 PLCs.

For information about the Modicon M580, refer to the chapter M580 Operating Modes (see Modicon M580, Hardware, Reference Manual).

Processing of Power Outage and Restoral of Modicon M340 PLCs

General

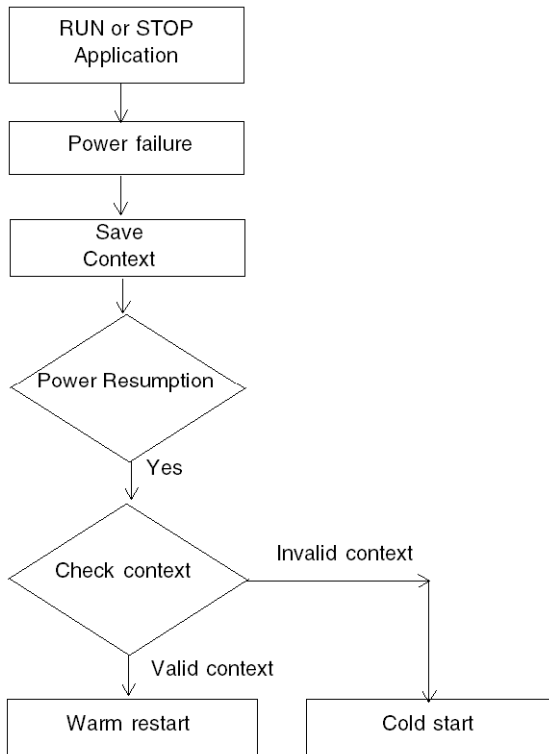
If the duration of the outage is less than the power supply filtering time, it has no effect on the program, which continues to run normally. If this is not the case, the program is interrupted and power restoration processing is activated.

Filtering time:

| PLC | Alternating Current | Direct Current |
|---------------|---------------------|----------------|
| BMX CPS 2000 | 10 ms | - |
| BMX CPS 3500 | | |
| BMX CPS 3540T | | |
| BMX CPS 4002 | | |
| BMX CPS 2010 | - | 1 ms |
| BMX CPS 3020 | | |

Illustration

The following illustration shows the different power cycle phases.



Operation

The following table describes the power outage processing phases.

| Phase | Description |
|-------|--|
| 1 | On power outage, the system saves the application context, the values of application variables, and the state of the system on internal Flash memory. |
| 2 | The system sets all the outputs into fallback state (state defined in configuration). |
| 3 | On power restoral, some actions and checks are done to verify if warm restart is available: <ul style="list-style-type: none"> restoring from internal Flash memory application context, verification with memory card (presence, application availability), verification that the application context is identical to the memory card context, If all checks are correct, a warm restart, page 141 is done, otherwise a cold start, page 137 is carried out. |

Processing on Cold Start for Modicon M340 PLCs

Cause of a Cold Start

The following table describes the different possible causes of a cold start.

| Causes | Startup characteristics |
|---|--|
| Loading of an application | Cold start forced in STOP |
| Restore application from memory card, when the application is different from the one in internal RAM | Cold start forced in STOP or RUN mode as defined in the configuration |
| Restore application from memory card, with Control Expert commands PLC > Project backup > | Cold start forced in STOP. The start in RUN mode as defined in the configuration is not taken into account |
| RESET button pressed on supply | Cold start forced in STOP or RUN mode as defined in the configuration |
| RESET button pressed on supply less than 500ms after a power down | Cold start forced in STOP or RUN mode as defined in the configuration |
| RESET button pressed on supply after a processor error, except in the case of a watchdog error | Cold start forced in STOP. The start in RUN mode as defined in the configuration is not taken into account |
| Initialization from Control Expert Forcing the system bit %S0 | Start in STOP or in RUN (retaining the operating mode in progress at downtime), initialization only of application |
| Restoral after power supply outage with loss of context | Cold start forced in STOP or RUN mode as defined in the configuration |

⚠ CAUTION**LOSS OF DATA ON APPLICATION TRANSFER**

Loading or transferring an application to the PLC typically involves initialization of unlocated variables.

To save the located variables:

- Avoid the initialization of the `%MWi` by unchecking **Initialize %MWi on cold start** in the **configuration screen** of the CPU.

It is necessary to assign a topological address to the data if the process requires keeping the current values of the data when transferring the application.

Failure to follow these instructions can result in injury or equipment damage.

⚠ CAUTION**LOSS OF DATA ON APPLICATION TRANSFER**

Do not press the RESET button on the power supply. Otherwise, `%MWi` is reset and initial values are loaded.

Failure to follow these instructions can result in injury or equipment damage.

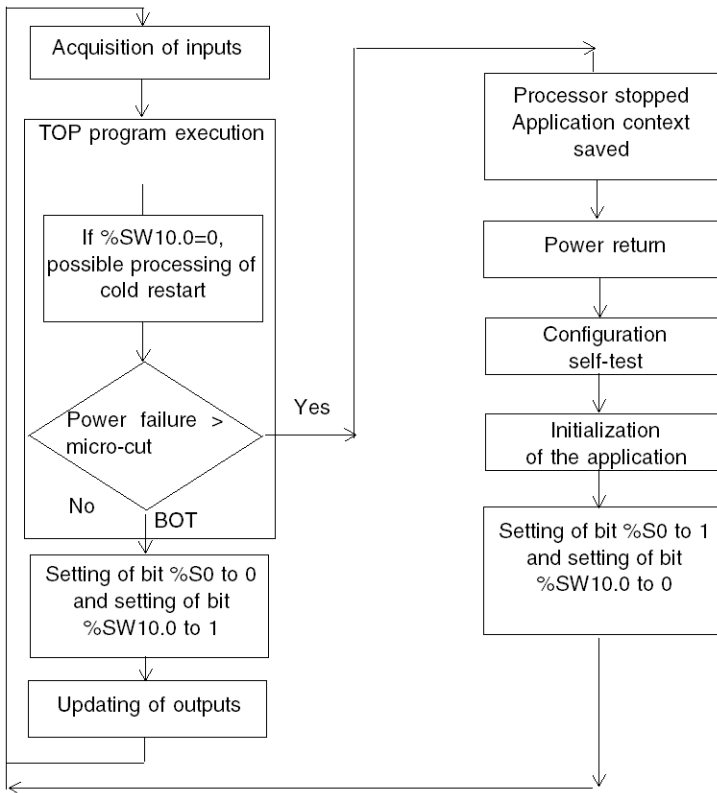
⚠ CAUTION**RISK OF LOSS OF APPLICATION**

If there is no memory Card in the PLC during a cold restart the application is lost.

Failure to follow these instructions can result in injury or equipment damage.

Illustration

The diagram below describes how a cold restart operates.



Operation

The table below describes the program execution restart phases on cold restart.

| Phase | Description |
|-------|---|
| 1 | <p>The startup is performed in RUN or in STOP depending on the status of the <code>Automatic start in RUN</code> parameter defined in the configuration or, if this is in use, depending on the state of the RUN/STOP input.</p> <p>Program execution is resumed at the start of the cycle.</p> |
| 2 | <p>The system carries out the following:</p> <ul style="list-style-type: none"> • Deactivating tasks, other than the master task, until the end of the first master task cycle. • Initializing data (bits, I/O image, words etc.) with the initial values defined in the data editor (value set to 0, if no other initial value has been defined). For %MW words, the values can be retrieved on cold restart if the two conditions are valid : <ul style="list-style-type: none"> ◦ the Initialize of %MW on cold restart option (see EcoStruxure™ Control Expert, Operating Modes) is unchecked in the processor's configuration screen, ◦ the internal flash memory has a valid backup (see %SW96 (see EcoStruxure™ Control Expert, System Bits and Words, Reference Manual)). <p>Note : If the number of %MW words exceeds the backup size (see the <code>memory structure of M340 PLCs</code>, page 123) during the save operation the remaining words are set to 0.</p> <ul style="list-style-type: none"> • Initializing elementary function blocks on the basis of initial data. • Initializing data declared in the DFBs: either to 0 or to the initial value declared in the DFB type. • Initializing system bits and words. • Positioning charts to initial steps. • Cancelling any forcing. • Initializing message and event queues. • Sending configuration parameters to all discrete input/output modules and application-specific modules. |
| 3 | <p>For this first restart cycle the system does the following:</p> <ul style="list-style-type: none"> • Relaunches the master task with the %S0 (cold restart) and %S13 (first cycle in RUN) bits set to 1, and the %SW10 word (detection of a cold restart during the first task cycle) is set to 0. • Resets the %S0 and %S13 bits to 0, and sets each bit of the word %SW10 to 1 at the end of this first cycle of the master task. • Activates the fast task and event processing at the end of the first cycle of the master task. |

Processing a cold start by program

It is advisable to test the bit %SW10.0 to detect a cold start and start processing specific to this cold start.

NOTE: It is possible to test the bit %S0, if the parameter `Automatic start in RUN` has been selected. If this is not the case, the PLC starts in STOP, the bit %S0 then switches to 1 on the first cycle after restart but is not visible to the program because it is not executed.

Output Changes

As soon as a power outage is detected, the outputs are set in the fallback position:

- either they are assigned the fallback value,
- or the current value is maintained,

depending on the choice made in the configuration.

After power restoral, the outputs remain at zero until they are updated by the task.

Processing on Warm Restart for Modicon M340 PLCs

Cause of a Warm Restart

A warm restart may be caused by a power restoral without loss of context.

⚠ CAUTION

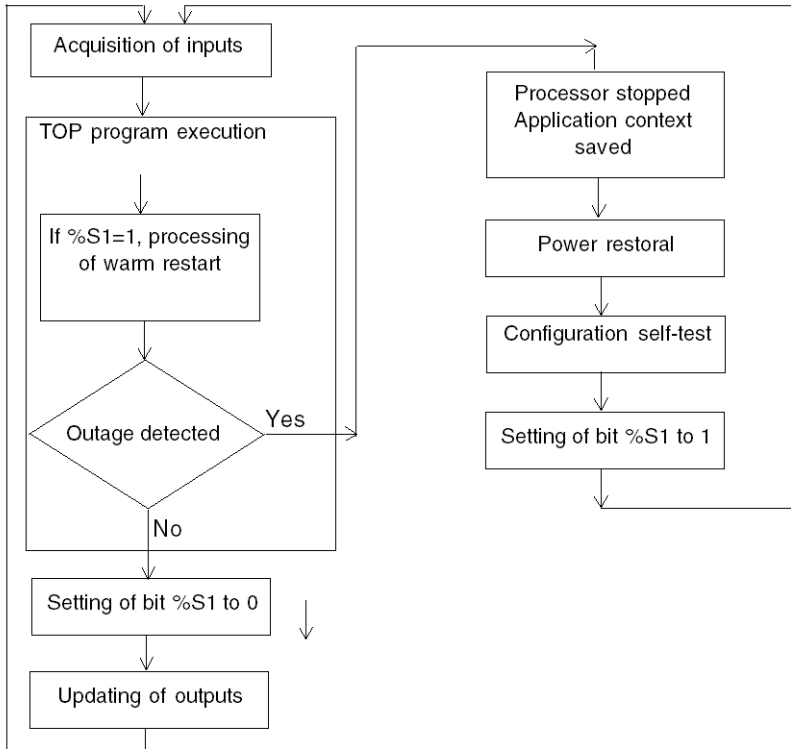
RISK OF LOSS OF APPLICATION

If there is no Memory Card in the PLC during a warm restart the application is lost.

Failure to follow these instructions can result in injury or equipment damage.

Illustration

The diagram below describes how a warm restart operates.



Operation

The table below describes the program execution restart phases on warm restart.

| Phase | Description |
|-------|--|
| 1 | Program execution doesn't resume from the element where the power outage occurred. The remaining program is discarded during the warm start. Each task will restart from the beginning. |
| 2 | At the end of the restart cycle, the system carries out the following: <ul style="list-style-type: none"> • restore the application's variable value, • set bit %S1 to 1, • the initialization of message and event queues, • the sending of configuration parameters to all discrete input/output and application-specific modules, • the deactivation of the fast task and event processing (until the end of the master task cycle). |
| 3 | The system performs a restart cycle during which it: <ul style="list-style-type: none"> • relaunches the master task from beginning of cycle, • resets bit %S1 to 0 at the end of this first master task cycle, • reactivates the fast task, event processing at the end of this first cycle of the master task. |

Processing a Warm Restart by Program

In the event of a warm restart, if you want the application to be processed in a particular way, you must write the corresponding program to test that %S1 is set to 1 at the start of the master task program.

SFC Warm start specific features

The Warm start on M340 PLCs is not considered as a real warm start by the CPU. SFC interpreter does not depend on tasks.

SFC publishes a memory area "ws_data" to the OS that contains SFC-section-specific data to be saved at a power fail. At the beginning of chart processing the currently active steps are saved to "ws_data" and processing is marked to be in "critical section". At the end of chart processing the "critical section" is unmarked.

If a power failure hits into "critical section" this could be detected if this state is active at the beginning (as the scan is aborted and MAST task is restarted from the beginning). In this case the workspace might be inconsistent and is restored from the saved data.

Additional information from SFCSTEP_STATE in located data area is used to reconstruct the state machine.

When a power failure occurs:

- during first scan %S1 =1 Mast is executed but Fast and Event tasks are not executed.

On power restoral:

- Clears chart, deregisters diagnostics, keeps set actions
- sets steps from saved area
- sets step times from SFCSTEP_STATE
- restores elapsed time for timed actions

NOTE: SFC interpreter is independent, if the transition is valid, the SFC chart evolves while %S1 is true.

Output Changes

As soon as a power outage is detected, the outputs are set in the fallback position:

- either they are assigned the fallback value,
- or the current value is maintained,

depending on the choice made in the configuration.

After power restoral, the outputs stay in security mode (equal to 0) until they are updated by a running task.

Automatic Start in RUN for Modicon M340 PLCs

Description

Automatic start in RUN is a processor configuration option. This option forces the PLC to start in RUN after a cold restart, page 137, except after an application has been loaded onto the PLC.

For Modicon M340 this option is not taken into account when the power supply RESET button is pressed after a processor error, except in the case of a watchdog error.

▲ WARNING

UNEXPECTED SYSTEM BEHAVIOR - UNEXPECTED PROCESS START

The following actions will trigger automatic start in RUN:

- Restoring the application from memory card,
- Unintentional or careless use of the reset button.

To avoid an unwanted restart when in RUN mode use:

- The RUN/STOP input on Modicon M340

Failure to follow these instructions can result in death, serious injury, or equipment damage.

Processing of State RAM on STOP Mode for Modicon M340 PLCs

General

With Modicon M340 firmware 2.4 or later, you can access the modules either via topological or State RAM addresses. Please also refer to *Memory Tab* (see EcoStruxure™ Control Expert, Operating Modes).

NOTE: The State RAM is refreshed in PLC RUN mode only.

The State RAM is **not** refreshed in PLC STOP mode.

Premium, Quantum PLCs Operating Modes

Subject of this Section

This section describes the operating modes of the Premium and Quantum PLCs.

Processing of Power Outage and Restoral for Premium/Quantum PLCs

General

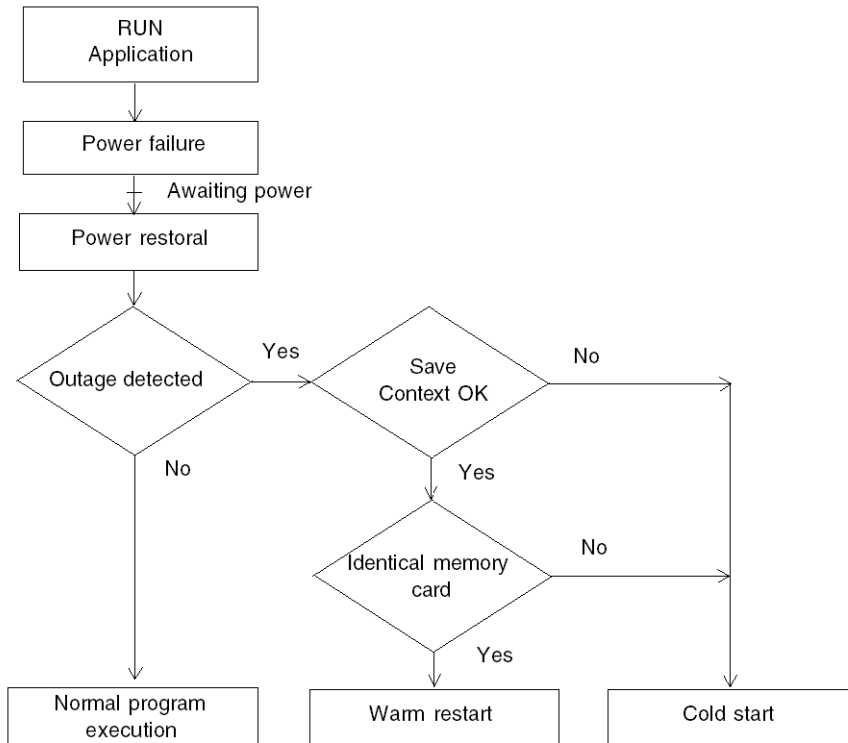
If the duration of the outage is less than the power supply filtering time, it has no effect on the program which continues to run normally. If this is not the case, the program is interrupted and power restoral processing is activated.

Filtering time:

| PLC | Alternating Current | Direct Current |
|---------|---------------------|----------------|
| Premium | 10ms | 1ms |
| Atrium | 30ms | - |
| Quantum | 10ms | 1ms |

Illustration

The illustration shows the different types of power restoral detected by the system.



Operation

The table below describes the power outage processing phases.

| Phase | Description |
|-------|--|
| 1 | On power outage the system stores the application context and the time of outage. |
| 2 | It sets all the outputs in the fallback state (state defined in configuration). |
| 3 | On power restoral, the saved context is compared to the current one, which defines the type of startup to be performed: <ul style="list-style-type: none"> if the application context has changed (i.e. loss of system context or new application), the PLC initializes the application: cold start, if the application context is the same, the PLC carries out a restart without initialization of data: warm restart. |

Power Outage on a Rack, Other than Rack 0

All the channels on this rack are seen as in error by the processor, but the other racks are not affected. The values of the inputs in error are no longer updated in the application memory and are reset to zero in a discrete input module, unless they have been forced, in which case they are maintained at the forcing value.

If the duration of the outage is less than the filtering time, it has no effect on the program which continues to run normally.

Processing on Cold Start for Premium/Quantum PLCs

Cause of a Cold Start

The following table describes the different possible causes of a cold start.

| Causes | Startup characteristics |
|--|--|
| Loading of an application | Cold start forced in STOP |
| RESET button pressed on processor (Premium) | Cold start forced in STOP or RUN mode as defined in the configuration |
| RESET button pressed on the processor after a processor or system error (Premium). | Cold start forced in STOP |
| Movement of handle or insertion/removal of a PCMCIA memory card | Cold start forced in STOP or RUN mode as defined in the configuration |
| Initialization from Control Expert Forcing the system bit %S0 | Start in STOP or in RUN (retaining the operating mode in progress at downtime), without initialization of discrete input/output and application-specific modules |
| Power restored after power supply outage with loss of context | Cold start forced in STOP or RUN mode as defined in the configuration |

⚠ CAUTION

LOSS OF DATA ON APPLICATION TRANSFER

Loading or transferring an application to the PLC typically involves initialization of unlocated variables.

To save located variables with Premium and Quantum PLCs:

- Save and restore %M and %MW by clicking **PLC > Transfer Data**.

For Premium PLCs:

- Avoid the initialization of %MW by un-checking **Initialize %MWi on cold start** in the **configuration screen** of the CPU.

For Quantum PLCs:

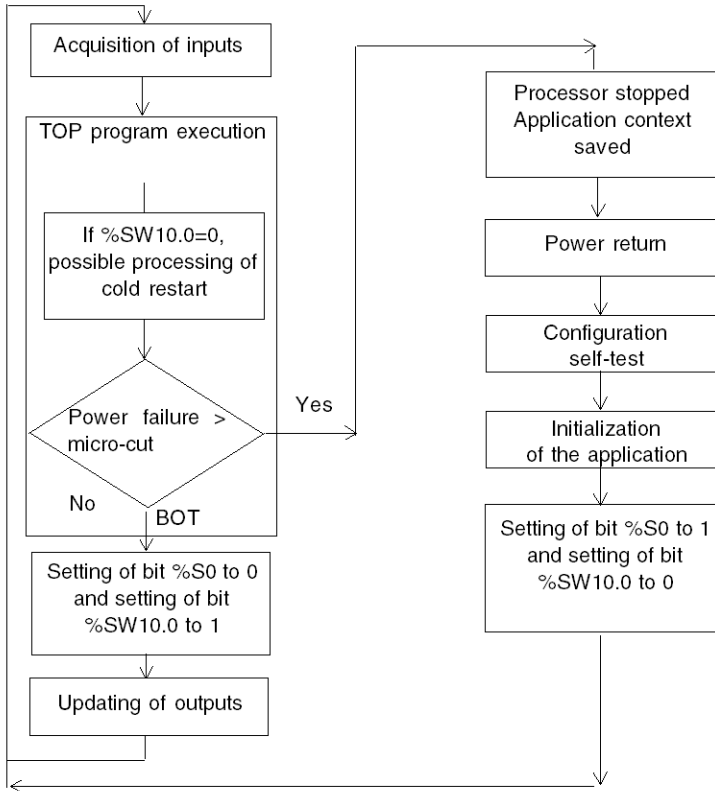
- Avoid the initialization of %MW by un-checking **Initialize %MWi** in the **configuration screen** of the CPU.

It is necessary to assign a topological address to the data if the process requires keeping the current values of the data when transferring the application.

Failure to follow these instructions can result in injury or equipment damage.

Illustration

The diagram below describes how a cold restart operates.



Operation

The table below describes the program execution restart phases on cold restart.

| Phase | Description |
|-------|--|
| 1 | <p>The startup is performed in RUN or in STOP depending on the status of the <code>Automatic start in RUN</code> parameter defined in the configuration or, if this is in use, depending on the state of the RUN/STOP input.</p> <p>Program execution is resumed at the start of the cycle.</p> |
| 2 | <p>The system carries out the following:</p> <ul style="list-style-type: none"> • the initialization of data (bits, I/O image, words etc.) with the initial values defined in the data editor (value set to 0, if no other initial value has been defined). For %MW words, the values can be retained on cold restart if the Reset of %MW on cold restart option is unchecked in the Configuration screen of the processor. NOTE: %MWi is not retained if a new program is loaded. • the initialization of elementary function blocks on the basis of initial data • the initialization of data declared in the DFBs: either to 0 or to the initial value declared in the DFB type • the initialization of system bits and words • the deactivation of tasks, other than the master task, until the end of the first master task cycle • the positioning of charts to initial steps • the cancellation of any forcing • the initialization of message and event queues • the sending of configuration parameters to all discrete input/output modules and application-specific modules |
| 3 | <p>For this first restart cycle the system does the following:</p> <ul style="list-style-type: none"> • relaunches the master task with the %S0 (cold restart) and %S13 (first cycle in RUN) bits set to 1, and the %SW10 word (detection of a cold restart during the first task cycle) is set to 0 • resets the %S0 and %S13 bits to 0, and sets each bit of the word %SW10 to 1 at the end of this first cycle of the master task • activates the fast task and event processing at the end of the first cycle of the master task |

Processing a Cold Start by Program

It is advisable to test the bit %SW10.0 to detect a cold start and start processing specific to this cold start.

NOTE: It is possible to test the bit %S0, if the parameter `Automatic start in RUN` has been selected. If this is not the case, the PLC starts in STOP, the bit %S0 then switches to 1 on the first cycle after restart but is not visible to the program because it is not executed.

Output Changes, for Premium and Atrium

As soon as a power outage is detected, the outputs are set in the fallback position:

- either they are assigned the fallback value, or
- the current value is maintained

depending on the choice made in the configuration.

After power restore, the outputs remain at zero until they are updated by the task.

Output Changes, for Quantum

As soon as a power outage is detected,

- the local outputs are set to zero
- the outputs of the remote or distributed extension racks are set in the fallback position

After power is restored, the outputs remain at zero until they are updated by the task.

NOTE: The behavior of forced outputs was changed between Modsoft/NxT/Concept and Control Expert.

With Modsoft/NxT/Concept, you cannot force outputs if the Quantum processor memory protection switch is set to "On".

With Control Expert, you can force outputs if the Quantum processor memory protection switch is set to "On".

With Modsoft/NxT/Concept, forced outputs retain their status after a cold start.

With Control Expert, forced outputs lose their status after a cold start.

⚠ CAUTION

UNEXPECTED APPLICATION BEHAVIOR - FORCED VARIABLES

Check your forced variables and memory protection switch when shifting between Modsoft/NxT/Concept and Control Expert.

Failure to follow these instructions can result in injury or equipment damage.

For Quantum 140 CPU 31•/41•/51••

These processors have a Flash EPROM memory of 1,435 KB which can be used to save the program and the initial values of variables.

When power is restored, you can choose the desired operating mode using the PLC MEM switch on the processor front panel. For detailed information on how this switch works, you can consult the Quantum manual (see Quantum using EcoStruxure™ Control Expert, Hardware, Reference Manual).

- **off position:** The application contained in this zone is automatically transferred to internal RAM when the PLC processor is powered up: cold restart of the application.
- **on position:** The application contained in this zone is not transferred to internal RAM: warm restart of the application.

Processing on Warm Restart for Premium/Quantum PLCs

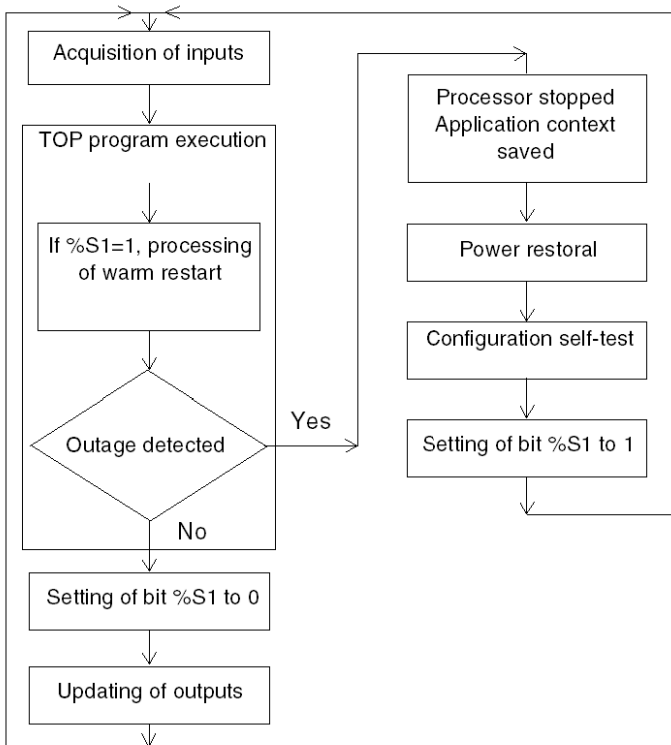
Cause of a Warm Restart

A warm restart may be caused:

- by a power restoral without loss of context
- by the system bit %S1 being set to 1 by the program
- by Control Expert from the terminal
- by pressing the RESET button of the power supply module of rack 0 (on Premium PLC)

Illustration

The diagram below describes how a warm restart operates.



Operation

The table below describes the program execution restart phases on warm restart.

| Phase | Description |
|-------|---|
| 1 | Program execution resumes starting from the element where the power outage occurred, without updating the outputs. |
| 2 | At the end of the restart cycle, the system carries out the following: <ul style="list-style-type: none"> • the initialization of message and event queues • the sending of configuration parameters to all discrete input/output and application-specific modules • the deactivation of the fast task and event processing (until the end of the master task cycle) |
| 3 | The system performs a restart cycle during which it: <ul style="list-style-type: none"> • re-acknowledges all the input modules • relaunches the master task with the bits %S1 (warm restart) set to 1 • resets bit %S1 to 0 at the end of this first master task cycle • reactivates the fast task, the auxiliary tasks and event processing at the end of this first cycle of the master task |

Processing a Warm Restart by Program

In the event of warm restart, if you want the application to be processed in a particular way, you must write the corresponding program conditional on the test that %S1 is set to 1 at the start of the master task program.

For Quantum PLCs, the switch on the front panel of the processor can be used to configure operating modes. For further details, see Quantum documentation (see Quantum using EcoStruxure™ Control Expert, Hardware, Reference Manual).

Output Changes, for Premium and Atrium

As soon as a power outage is detected, the outputs are set in the fallback position:

- either they are assigned the fallback value, or
- the current value is maintained.

depending on the choice made in the configuration.

After power restoral, the outputs remain in the fallback position until they are updated by the task.

NOTE: after a power on while the CPU is not started, outputs are in security mode state (equal to 0). After the CPU start, if the module didn't stay powered on, the maintain state is lost and the output stay in state 0.

Output Changes, for Quantum

As soon as a power outage is detected:

- the local outputs are set to zero
- the outputs of the remote or distributed extension racks are set in the fallback position

After power restoral, the outputs remain in the fallback position until they are updated by the task.

Output Changes, for Extension Rack

If power outage occurs on rack where CPU is located:

- Fallback state as soon as CPU loss is detected
- Security state during I/O configuration
- State calculated by CPU after the first run of the task driving this output

After power is restored, the outputs remain in the fallback position until they are updated by the task

Automatic Start in RUN for Premium/Quantum

Description

Automatic start in RUN is a processor configuration option. This option forces the PLC to start in RUN after a cold restart, page 147, except after an application has been loaded onto the PLC.

For Quantum PLCs, automatic start in RUN also depends on the position of the switch on the front panel of the processor. For more details, refer to the Quantum documentation (see Quantum using EcoStruxure™ Control Expert, Hardware, Reference Manual).

▲ WARNING

UNEXPECTED SYSTEM BEHAVIOR - UNEXPECTED PROCESS START

The following actions will trigger "automatic start in RUN":

- Inserting the PCMCIA card when the PLC is powered up (Premium, Quantum),
- Replacing the processor while powered up (Premium, Quantum),
- Unintentional or careless use of the reset button,
- If the battery is found to be defective in the event of a power outage (Premium, Quantum).

To avoid an unwanted restart when in RUN mode:

- We strongly recommend to use the RUN/STOP input on Premium PLCs or the switch on the front of the panel of the processor for Quantum PLCs
- We strongly recommend **not** to use memorized inputs as RUN/STOP input of the PLC.

Failure to follow these instructions can result in death, serious injury, or equipment damage.

PLC HALT Mode

Subject of this Section

This section describes an overview of the HALT mode.

PLC HALT Mode

At a Glance

The following actions switches the PLC to HALT mode:

- using the HALT instruction
- watchdog overflow
- Program execution error (division by zero, overflow, etc.) if the bit %S78 (see EcoStruxure™ Control Expert, System Bits and Words, Reference Manual) is set to 1.

Precaution

⚠ WARNING

UNEXPECTED APPLICATION BEHAVIOR

When the PLC is in Halt, all tasks are stopped. Check the behavior of the associated I/Os to ensure that the consequences of the PLC Halt on the application are acceptable.

Failure to follow these instructions can result in death, serious injury, or equipment damage.

Data Description

What's in This Part

| | |
|-------------------------------|-----|
| General Overview of Data..... | 159 |
| Data Types | 166 |
| Data Instances | 226 |
| Data References..... | 241 |

In This Part

This part describes the different data types that can be used in a project, and how to implement them.

General Overview of Data

What's in This Chapter

| | |
|---|-----|
| General..... | 159 |
| General Overview of the Data Type Families..... | 160 |
| Overview of Data Instances..... | 162 |
| Overview of the Data References..... | 163 |
| Syntax Rules for Type\Instance Names..... | 164 |

Subject of this Chapter

This chapter provides a general overview of:

- the different data types
- the data instances
- the data references

General

Introduction

A **data** item designates an object which can be instantiated such as:

- a variable,
- a function block.

Data is defined in three phases. These are:

- the **data types** phase, which specifies the following:
 - its category,
 - its format.
- the **data instances** phase, which defines its storage location and property, which is:
 - located, or
 - unlocated.
- the **data references** phase, which defines its means of access:
 - by immediate value,
 - by name,
 - by address.

Illustration

The following are the three phases that characterize the data:



Instantiating a data item consists in allocating it a memory slot according to its type.

Referencing a data item consists in defining a reference for it (name, address, etc.) allowing it to be accessed in the memory.

General Overview of the Data Type Families

Introduction

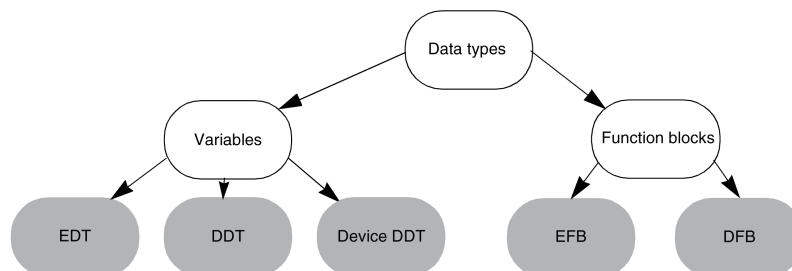
A **data type** is a piece of software information which specifies for a data item:

- its structure
- its format
- a list of its attributes
- its behavior

These properties are shared by all instances of the data type.

Illustration

The data type families are filed in different categories (dark gray).



Definitions

Data type families and their definitions.

| Family | Definition |
|------------|--|
| EDT | Elementary data types, such as: <ul style="list-style-type: none"> • Bool • Int • Byte • Word • Dword • etc. |
| DDT | Derived Data Types, such as: <ul style="list-style-type: none"> • Arrays, which contain elements of the same type: <ul style="list-style-type: none"> ◦ Bool tables (EDT tables) ◦ EBool tables (Device DDT Arrays) ◦ tables of tables (DDT tables) ◦ tables of structures (DDT tables) • structures, which contain elements of the different types: <ul style="list-style-type: none"> ◦ Bool structures, Word structures, etc. (EDT structures) ◦ EBool tables (Device DDT structure) ◦ structures of tables, structures of structures, structures of tables/structures (DDT structures) ◦ Bool structures, table structures, etc. (EDT and DDT structures) ◦ structures concerning input/output data (IODDT structures) |
| Device DDT | Device Derived Data Types, such as: <ul style="list-style-type: none"> • tables, which contain elements of the same type: <ul style="list-style-type: none"> ◦ Bool tables (EDT tables) ◦ tables of tables (DDT tables) ◦ tables of structures (DDT tables) • structures, which contain elements of the different types: <ul style="list-style-type: none"> ◦ Bool structures, Word structures, etc. (EDT structures) ◦ structures of tables, structures of structures, structures of tables/structures (DDT structures) ◦ Bool structures, table structures, etc. (EDT and DDT structures) ◦ structures concerning input/output data ◦ Structures containing variables that restore the status properties of an action or transition of a Sequential Function Chart |

| Family | Definition |
|--------|---|
| EFB | Elementary Function Blocks written in C language. These comprise: <ul style="list-style-type: none">• input variables• internal variables• output variables• a processing algorithm |
| DFB | Derived Function Blocks written in automation languages (Structured Text, Instruction List, etc.). These comprise: <ul style="list-style-type: none">• input variables• internal variables• output variables• a processing algorithm |

Overview of Data Instances

Introduction

A **data instance** is an individual functional entity, which has all the characteristics of the data type to which it belongs.

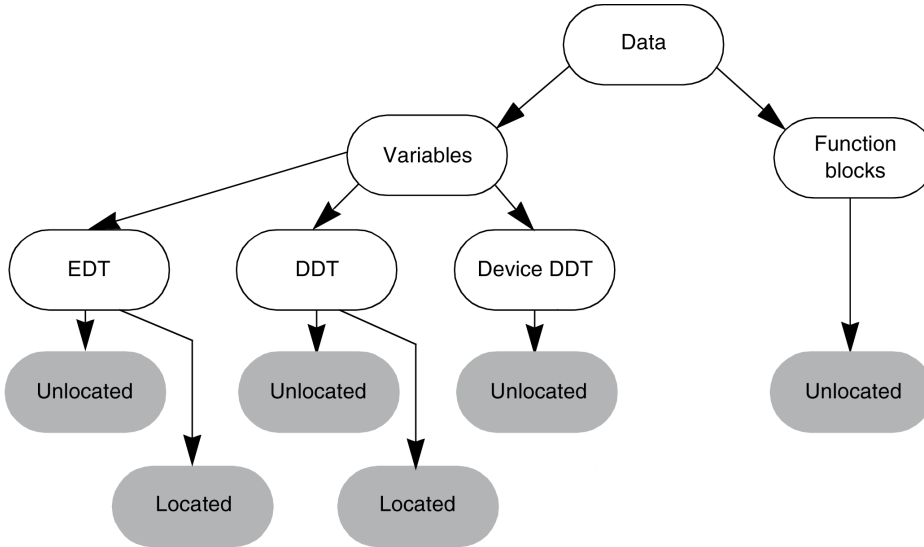
One or more instances can belong to a data type.

The data instance can have a memory allocation that is:

- unlocated or
- located

Illustration

Memory allocation of instances (dark gray) belonging to the different types.



Definitions

Definition of the memory allocations of data instances.

| Data instance | Definition |
|---------------|--|
| Unlocated | <p>The memory slot of the instance is automatically allocated by the system and can change for each generation of the application.</p> <p>The instance is located by a name (symbol) chosen by the user.</p> |
| Located | <p>The memory slot of the instance is fixed, predefined and never changes.</p> <p>The instance is located by a name (symbol) chosen by the user and a topological address defined by the manufacturer, or by the topological address of the manufacturer only.</p> |

Overview of the Data References

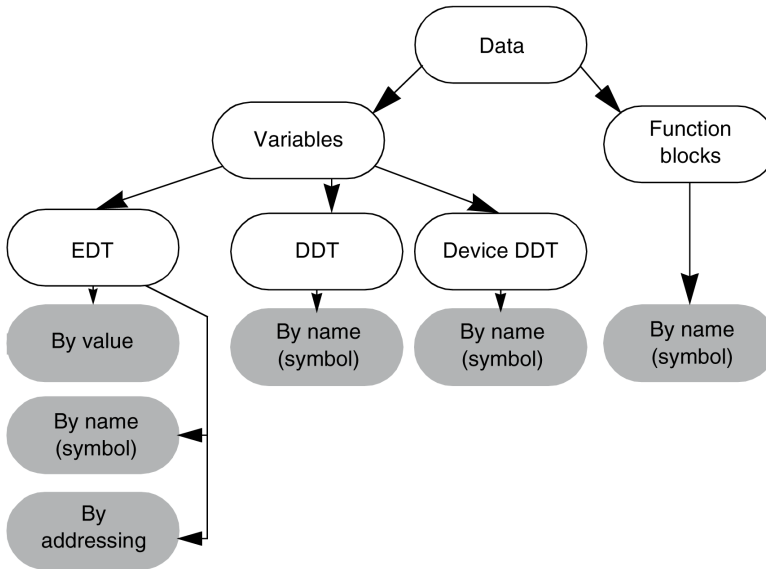
Introduction

A **data reference** allows the user to access the instance of this data either by:

- immediate value, true only for data of type EDT
- address settings, true only for data of type EDT
- name (symbol), true for all EDT, DDT, EFB, DFB data types, as well as for SFC objects

Illustration

Possible data references according to data type (dark gray).



Syntax Rules for Type\Instance Names

Introduction

The syntax of names of types and variables can be written up with **Standard**, **Extended**, or **Unicode** character set. The option **Character set** is configured in **Tools > Project Settings > Variables**.

The **Character set** setting is used for names entered into the application concerns:

- DFB (Derived Function Block) user function blocks or DDT (Derived data type)
- the internal elements composing a DFB or a DDT
- the data instances

Character set Options

With **Standard** option selected:

- The application is compliant with the IEC standard
- The names entered are strings made up of alphanumeric characters and the Underscore character.
 - the first character of the name is an alphabetic character or an Underscore
 - two Underscore characters cannot be used consecutively

With **Extended** option selected:

- The user has a certain degree of flexibility, but the application is not compliant with the IEC standard
- Extra characters can be used:
 - characters corresponding to ASCII codes 192 to 223 (except for code 215)
 - characters corresponding to ASCII codes 224 to 255 (except for code 247)
- the first character of the name is an **alphanumeric** character or an Underscore
- Underscore characters **can be** used consecutively

Select **Unicode** to create elements in not-roman alphabets like Chinese.

Data Types

What's in This Chapter

| | |
|--|-----|
| Elementary Data Types (EDT) in Binary Format | 166 |
| Elementary Data Types (EDT) in BCD Format..... | 177 |
| Elementary Data Types (EDT) in Real Format..... | 181 |
| Elementary Data Types (EDT) in Character String Format | 187 |
| Elementary Data Types (EDT) in Bit String Format | 189 |
| Derived Data Types (DDT/IODDT/Device DDT) | 192 |
| Function Block Data Types (DFB\EFB) | 208 |
| Generic Data Types (GDT) | 214 |
| Data Types Belonging to Sequential Function Charts (SFC)..... | 216 |
| Compatibility Between Data Types | 218 |
| Reference Data Type Declarations..... | 222 |

Subject of this Chapter

This chapter describes all the data types that can be used in an application.

Elementary Data Types (EDT) in Binary Format

Subject of this Section

This section describes Binary format data types. These are:

- Boolean types
- Integer types
- Time types

Overview of Data Types in Binary Format

Introduction

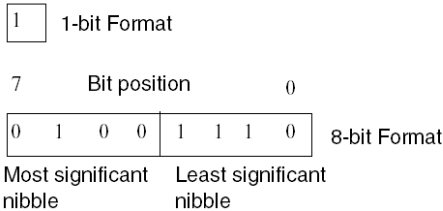
The data types in Binary format belong to the EDT (Elementary data type) family, which includes **single** rather than derived data types (tables, structures, function blocks).

Reminder Concerning Binary Format

A data item in binary format is made up of one or more bits, where each of these is represented by one of the base 2 figures (**0** or **1**).

The scale of the data item depends on the number of bit(s) of which it is made.

Example:



A data item can be:

- signed. Here the highest ranking bit is the sign bit:
 - 0 indicates a positive value
 - 1 indicates a negative value

The range of values is:

$$[-2^{(Bits-1)}, 2^{(Bits-1)} - 1]$$

- unsigned. Here all the bits represent the value

The range of values is:

$$[0, 2^{Bits} - 1]$$

Bits=number of bits (format).

Data Types in Binary Format

List of data types:

| Type | Designation | Format (bits) | Default value |
|-------|---|---------------|---------------|
| BOOL | Boolean | 8 | 0=(False) |
| EBOOL | Boolean with forcing and edge detection | 8 | 0=(False) |
| INT | Integer | 16 | 0 |

| Type | Designation | Format (bits) | Default value |
|-------|-------------------------|---------------|---------------|
| DINT | Double integer | 32 | 0 |
| UINT | Unsigned integer | 16 | 0 |
| UDINT | Unsigned double integer | 32 | 0 |
| TIME | Unsigned double integer | 32 | T=0s |

Boolean Types

At a Glance

There are three types of boolean:

- `BOOL`, page 168 type, which contains only the value `FALSE (=0)` or `TRUE (=1)`.
- `EBOOL`, page 169 type, which contains the value `FALSE (=0)` or `TRUE (=1)` but also information concerning the management of falling or rising edges and forcing.
- `ANY_BOOL`, page 171 type, only declared as a referenced data type that combines `BOOL` and `EBOOL` types.

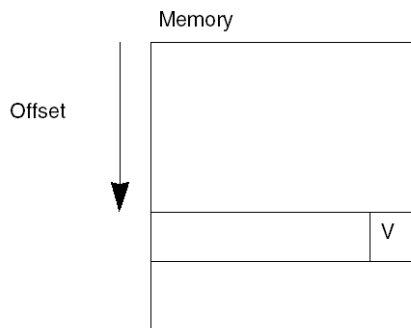
Principle of the `BOOL` Type

This type takes up one memory byte, but the value is only stored in 1 bit.

The default value for this type is `FALSE (=0)`.

It is accessible via an address containing the offset of the corresponding byte:

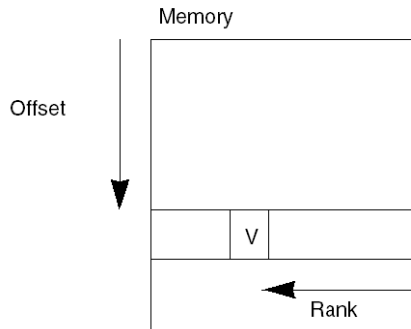
Address settings:



In the case **of the word extracted bit**, it is accessible via an address containing the following information:

- an offset of the corresponding byte
- the rank defining its position in the word

Address settings:



Principle of the EBOOL Type

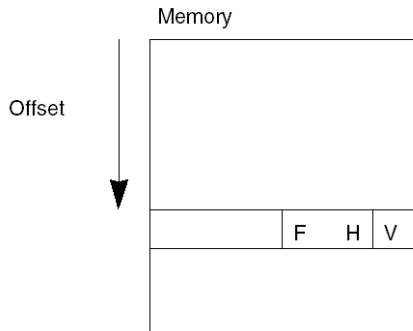
This type takes up one memory byte which contains:

- the bit for the value (V),
- the history bit (H) for managing rising or falling edges. Each time the object status changes, the value is copied to this bit,
- the bit containing the forcing status (F). Equal to 0 if the object is not forced and equal to 1 if the object is forced.

The default value for the bits associated with the EBOOL type is FALSE (=0).

It is accessible via an address specifying the offset of the corresponding byte:

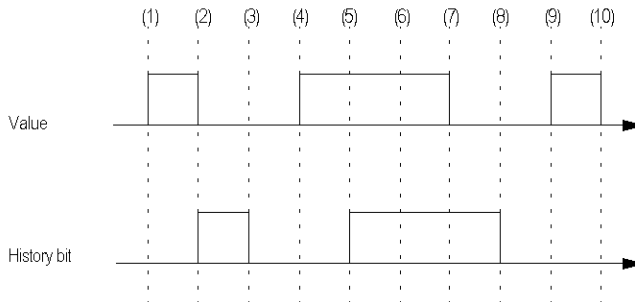
Address settings:



EBOOL Type Historical Trend Diagram

The trend diagram below shows the main statuses of the value and history bits associated with the EBOOL type.

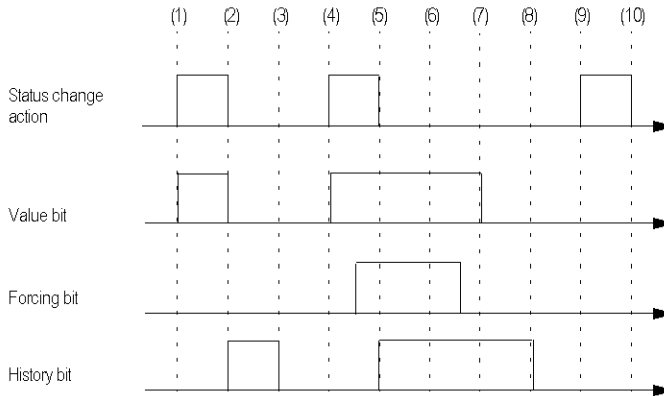
The rising edges of the value bit (1, 4) are copied to the history bit in the next PLC cycle (2, 5). The falling edges of the value bit (2, 7) are copied to the history bit of the next PLC cycle (3, 8).



EBOOL Type Trend Diagram and Forcing

The trend diagram below shows the main statuses of the value, history, and forcing bits associated with the EBOOL type.

The rising edges of the value bit (1, 4) are copied to the history bit in the next PLC cycle (2, 5). The falling edges of the value bit (2, 7) are copied to the history bit in the next PLC cycle (3, 8). Between (4 and 5), the forcing bit equals 1 while the value and history bits remain at 1.



Principle of the ANY_BOOL Type

The ANY_BOOL type can be used by supervision tools (a SCADA for example) to reserve variables declared as generic data type. The generic data type is the element shared with Control Expert.

An ANY_BOOL type variable is declared as a reference, using the REF_TO keyword. More details on referencing and dereferencing are provided in the topic on Reference Data Type Declarations, page 222.

NOTE: Implicit conversion is allowed on dereferenced ANY_BOOL type variable (BOOL_TO_*).

Usage limitation of ANY_BOOL type:

- The ANY_BOOL type cannot be used to declare a variable in Control Expert application. A variable is declared using a reference to ANY_BOOL type with keyword REF_TO.
- Referencing REF_TO_ANY_BOOL is not allowed in program.

MyRefToAnyBoolVar := REF(MyVar); is not allowed (whatever MyVar is: BOOL or EBOOL).

- In an EF or EFB, ANY_BOOL type cannot be used to declare a parameter or variable, even as a reference with keyword REF_TO.
- To reference an EBOOL, only the edge history is managed. The forcing functionality is not managed by the ANY_BOOL type when referencing an EBOOL.
- In a SCADA system, the ANY_BOOL type variable is the shared element, the data dictionary provides the final type of the ANY_BOOL reference (BOOL or EBOOL).

- A reference to a reference is not supported.
Cascading dereference is not supported (for example, `MyAnyBool1^MyAnyBool2^ .xy` is not supported).

Platform: `ANY_BOOL` type is used on the following platforms:

- Modicon M580 (OS version ≥ V2.00)
- Modicon Quantum 140CPU6•••• (OS version ≥ V3.30)
- Modicon M340 (OS version ≥ V2.70)

Time stamping: An `ANY_BOOL` reference variable can only be time stamped in system time stamping (see System Time Stamping, User Guide) mode if the referenced variable is a constant (`IsConstant` attribute enabled). The referenced variable can be associated to:

- A BMX ERT 1604 T source.
- A BMX CRA 312 10 source.
- A BME CRA 312 10 source.
- A Modicon M580 CPU source (OS version ≥ V2.00).
- A topological variable (for example `%M100`).

PLC Variables Belonging to Boolean Types

List of variables

| Variable | Type |
|--------------------|-------|
| Internal bit | EBOOL |
| System bit | BOOL |
| Word extracted bit | BOOL |
| %I inputs | |
| Module error bit | BOOL |
| Channel error bit | BOOL |
| Input bit | EBOOL |
| %Q outputs | |
| Output bit | EBOOL |

Compatibility Between BOOL and EBOOL

The operations authorized between these two types of variables are:

- value copying

- address copying

Copies between types

| | BOOL destination | EBOOL destination |
|---------------------|-------------------------|--------------------------|
| BOOL source | Yes | Yes |
| EBOOL source | Yes | Yes |

Compatibility between the parameters of elementary functions (EF)

| Effective parameter (external to EF) | Formal BOOL parameter (internal to EF) | Formal EBOOL parameter (internal to EF) |
|---|---|--|
| BOOL | Yes | No |
| EBOOL | In ->Yes In-Out ->No Out ->Yes | Yes |

Compatibility between the parameters of block functions (EFB\DFB)

| Effective parameter (external to FB) | Formal BOOL parameter (internal to FB) | Formal EBOOL parameter (internal to FB) |
|---|---|--|
| BOOL | Yes | In ->Yes In-Out ->No Out -> Yes |
| EBOOL | In ->Yes In-Out ->No Out -> Yes | Yes |

Compatibility between array variables

| | ARRAY[i..j] OF BOOL destination | ARRAY[i..j] OF EBOOL destination |
|------------------------------------|--|---|
| ARRAY[i..j] OF BOOL source | Yes | No |
| ARRAY[i..j] OF EBOOL source | No | Yes |

Compatibility between static variables

| | BOOL (%MW:xi) direct addressing | EBOOL (%Mi) direct addressing |
|--|--|--------------------------------------|
| BOOL (Var:BOOL) declared variable | Yes | No |
| EBOOL (Var:EBOOL) declared variable | No | Yes |

Compatibility

EBOOL data types follow the rules below:

- An EBOOL type variable cannot be passed as a BOOL type input/output parameter.
- EBOOL arrays cannot be passed as ANY type parameters of an FFB.
- BOOL and EBOOL arrays are not compatible for instructing assignment (same rule as for FFB parameters).
- On Quantum:
 - EBOOL type located variables cannot be passed as EBOOL type input/output parameters.
 - EBOOL arrays cannot be passed as parameters of a DFB.

Integer Types

At a Glance

Integer types are used to represent a value in different bases. These are:

- base 10 (decimal) by default. Here the value is signed or unsigned depending on the integer type
- base 2 (binary). Here the value is unsigned and the prefix is **2#**
- base 8 (octal). Here the value is unsigned and the prefix is **8#**
- base 16 (hexadecimal). Here the value is unsigned and the prefix is **16#**

NOTE: In decimal representation, if the chosen type is signed, the value can be preceded by the + sign or - sign (the + sign is optional).

Integer Type (INT)

Signed type with a 16-bit format.

This table shows the range in each base.

| Base | from... | to... |
|-------------|--------------------|--------------------|
| Decimal | -32768 | 32767 |
| Binary | 2#1000000000000000 | 2#0111111111111111 |
| Octal | 8#100000 | 8#077777 |
| Hexadecimal | 16#8000 | 16#7FFF |

Double Integer Type (DINT)

Signed type with a 32-bit format.

This table shows the range in each base.

| Base | from... | to... |
|-------------|------------------------------------|------------------------------------|
| Decimal | -2147483648 | 2147483647 |
| Binary | 2#10000000000000000000000000000000 | 2#01111111111111111111111111111111 |
| Octal | 8#20000000000 | 8#1777777777 |
| Hexadecimal | 16#80000000 | 16#7FFFFFFF |

Unsigned Integer Type (UINT)

Unsigned type with a 16-bit format.

This table shows the range in each base.

| Base | from... | to... |
|-------------|---------|--------------------|
| Decimal | 0 | 65535 |
| Binary | 2#0 | 2#1111111111111111 |
| Octal | 8#0 | 8#177777 |
| Hexadecimal | 16#0 | 16#FFFF |

Unsigned Double Integer Type (UDINT)

Unsigned type with a 32-bit format.

This table shows the range in each base.

Elementary Data Types (EDT) in BCD Format

Subject of this section

This section describes BCD format (Binary Coded Decimal) data types. These are:

- Date type
- Time of Day type (TOD)
- Date and Time (DT) type

Overview of Data Types in BCD Format

Introduction

The data types in BCD format belong to the EDT (Elementary data type) family, which includes **single** rather than derived data types (tables, structures, function blocks).

Reminder Concerning BCD Format

The Binary Coded Decimal (BCD) format is used to represent decimal numbers **between 0 and 9** using a group of four bits (half-byte).

In this format, the four bits used to code the decimal numbers have a range of unused combinations.

Correspondence table:

| Decimal | Binary |
|---------|---------------|
| 0 | 0000 |
| 1 | 0001 |
| 2 | 0010 |
| 3 | 0011 |
| 4 | 0100 |
| 5 | 0101 |
| 6 | 0110 |
| 7 | 0111 |
| 8 | 1000 |
| 9 | 1001 |
| | 1010 (unused) |

| Decimal | Binary |
|---------|---------------|
| | 1011 (unused) |
| | 1100 (unused) |
| | 1101 (unused) |
| | 1110 (unused) |
| | 1111 (unused) |

Example of coding using a 16 bit format:

| | | | | |
|----------------------|------|------|------|------|
| Decimal value | 2 | 4 | 5 | 0 |
| 2450 | | | | |
| Binary value | 0010 | 0100 | 0101 | 0000 |

Example of coding using a 32 bit format:

| | | | | | | | | |
|----------------------|------|------|------|------|------|------|------|------|
| Decimal value | 7 | 8 | 9 | 9 | 3 | 0 | 1 | 6 |
| 78993016 | | | | | | | | |
| Binary value | 0111 | 1000 | 1001 | 1001 | 0011 | 0000 | 0001 | 0110 |

Data Types in BCD Format

Three data types:

| Type | Designation | Scale (bits) | Default value |
|---------------|---------------|--------------|------------------------|
| DATE | Date | 32 | D#1990-01-01 |
| TIME_OF_DAY | Time of day | 32 | TOD#00:00:00 |
| DATE_AND_TIME | Date and Time | 64 | DT#1990-01-01-00:00:00 |

The Date Type

At a Glance

The **Date** type in 32 bit format contains the following information:

- the year coded in a 16-bit field (4 most significant half-bytes)
- the month coded in an 8-bit field (2 half bytes)

- the day coded in an 8-bit field (2 least significant half bytes)

Representation in BCD format of the date 2001-09-20:

| Year (2001) | Month (09) | Day (20) |
|---------------------|------------|-----------|
| 0010 0000 0000 0001 | 0000 1001 | 0010 0000 |

Syntax Rules

The **Date** type is entered as follows: **D#<Year>-<Month>-<Day>**

This table shows the lower/upper limits in each field.

| Field | Limits | Comment |
|-------|-------------|---|
| Year | [1990,2099] | |
| Month | [01,12] | The left 0 is always displayed, but can be omitted at the time of entry |
| Day | [01,31] | For the months 01\03\05\07\08\10\12 |
| | [01,30] | For the months 04\06\09\11 |
| | [01,29] | For the month 02 (leap years) |
| | [01,28] | For the month 02 (non leap years) |

Example:

| Entry | Comments |
|----------------------|--|
| D# 2001-1-1 | The left 0 of the month and the day can be omitted |
| d# 1990-02-02 | The prefix can be written in lower case |

The Time of Day (TOD) Type

At a Glance

The **Time of Day** type coded in 32 bit format contains the following information:

- the hour coded in an 8-bit field (2 most significant half-bytes)
- the minutes coded in an 8-bit field (2 half bytes)
- the seconds coded in an 8-bit field (2 half bytes)

NOTE: The 8 least significant bits are unused.

Representation in BCD format of the time of day 13:25:47:

| Hour (13) | Minutes (25) | Seconds (47) | Least significant byte |
|-----------|--------------|--------------|------------------------|
| 0001 0011 | 0010 0101 | 0100 0111 | Unused |

Syntax Rules

The Time of Day type is entered as follows: **TOD#**<Hour>:<Minutes>:<Seconds>

This table shows the lower/upper limits in each field.

| Field | Limits | Comment |
|--------|---------|---|
| Hour | [00,23] | The left 0 is always displayed, but can be omitted at the time of entry |
| Minute | [00,59] | The left 0 is always displayed, but can be omitted at the time of entry |
| Second | [00,59] | The left 0 is always displayed, but can be omitted at the time of entry |

Example:

| Entry | Comment |
|----------------------|--|
| TOD# 1:59:0 | The left 0 of the hours and seconds can be omitted |
| tod# 23:10:59 | The prefix can be written in lower case |
| Tod# 0:0:0 | The prefix can be mixed (lower\upper case) |

The Date and Time (DT) Type

At a Glance

The **Date and Time** type coded in 64 bit format contains the following information:

- The year coded in a 16-bit field (4 most significant half-bytes)
- the month coded in an 8-bit field (2 half bytes)
- the day coded in an 8-bit field (2 half bytes)
- the hour coded in an 8-bit field (2 half bytes)
- the minutes coded in an 8-bit field (2 half bytes)
- the seconds coded in an 8-bit field (2 half bytes)

NOTE: The 8 least significant bits are unused.

Example: Representation in BCD format of the date and Time 2000-09-20:13:25:47.

| Year (2000) | Month (09) | Day (20) | Hour (13) | Minute (25) | Seconds (47) | Least significant byte |
|------------------------|--------------|-----------|-----------|--------------|--------------|------------------------|
| 0010 0000 0000 0000 | 0000 1001 | 0010 0000 | 0001 0011 | 0010 0101 | 0100 0111 | Unused |

Syntax Rules

The **Date and Time** type is entered as follows:

DT#<Year>-<Month>-<Day>-<Hour>:<Minutes>:<Seconds>

This table shows the lower/upper limits in each field.

| Field | Limits | Comment |
|--------|-------------|---|
| Year | [1990,2099] | |
| Month | [01,12] | The left 0 is always displayed, but can be omitted during entry |
| Day | [01,31] | For the months 01\03\05\07\08\10\12 |
| | [01,30] | For the months 04\06\09\11 |
| | [01,29] | For the month 02 (leap years) |
| | [01,28] | For the month 02 (non leap years) |
| Hour | [00,23] | The left 0 is always displayed, but can be omitted during entry |
| Minute | [00,59] | The left 0 is always displayed, but can be omitted during entry |
| Second | [00,59] | The left 0 is always displayed, but can be omitted during entry |

Example:

| Entry | Comment |
|--------------------------------|--|
| DT# 2000-1-10-0:40:0 | The left 0 of the month\hour\second can be omitted |
| dt# 1999-12-31-23:59:59 | The prefix can be written in lower case |
| Dt# 1990-10-2-12:02:30 | The prefix can be mixed (lower\upper case) |

Elementary Data Types (EDT) in Real Format

Subject of this Section

This section describes data types in Real format.

Presentation of the Real Data Type

Introduction

The data types in Real format belong to the EDT (Elementary data type) family, which includes **single** rather than derived data types (tables, structures, function blocks).

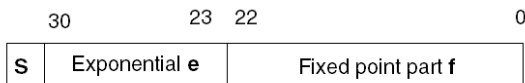
Reminder Concerning Real Format

The Real format (floating point in ANSI/IEEE 754 standard) is coded in 32 bit format which corresponds to the single decimal point floating numbers.

The 32 bits representing the floating point value are organized in three distinct fields. These are:

- **S**, the sign bit which can have the value:
 - 0, for a positive floating point number
 - 1, for a negative floating point number
- **e**, the exponential coded in an 8 bit field (integer in binary format)
- **f**, the fixed-point part coded in a 23 bit field (integer in binary format)

Representation:



The value of the fixed-point part (Mantissa) is between $[0, 1[$, and is calculated using the following formula.

$$F = 2^{-23} * M$$

Number Types that Can Be Represented

These are the numbers which are:

- normalized
- denormalized
- of infinite values
- with values +0 and -0

This table gives the values in the different fields according to number type.

| e | f | S | Number type |
|----------|-----------------------|----------------|-------------------|
|]0, 255[|]0, 1[| 0 or 1 | normalized |
| 0 |]0, 1[| near (1.4E-45) | denormalized DEN |
| 255 | 0 | 0 | + infinity (INF) |
| 255 | 0 | 1 | - infinity (-INF) |
| 255 |]0, 1[and bit 22 = 0 | 0 or 1 | SNAN |
| 255 |]0, 1[and bit 22 = 1 | 0 or 1 | QNAN |
| 0 | 0 | 0 | +0 |
| 0 | 0 | 1 | -0 |

NOTE: Standard IEC 559 defines two classes of NAN (not a number): QNAN and SNAN.

- QNAN: is a NAN whose bit 22 is set to 1
- SNAN: is a NAN whose bit 22 is set to 0

They behave as follows:

- QNAN do not trigger errors when they appear in operands of a function or an expression.
- SNAN trigger an error when they appear in operands of a function or an arithmetic expression (See %SW17 (see EcoStruxure™ Control Expert, System Bits and Words, Reference Manual) and %S18 (see EcoStruxure™ Control Expert, System Bits and Words, Reference Manual)).

This table gives the calculation formula of the value of the floating-point number:

| Floating-point number | Value |
|-----------------------|--|
| Normalized | $F = (-1)^S \times 2^{(e-127)} \times \left(1 + \frac{M}{2^{23}}\right)$ |
| Denormalized (DEN) | $F = (-1)^S \times 2^{-126} \times \left(\frac{M}{2^{23}}\right)$ |

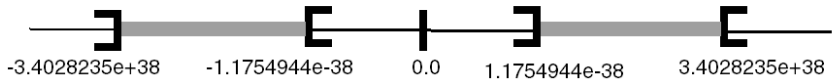
NOTE: A real number between $-1.1754944e-38$ and $1.1754944e-38$ is a denormalized DEN. When an operand is a DEN, the result is not guaranteed. The bits %SW17 (see EcoStruxure™ Control Expert, System Bits and Words, Reference Manual) and %S18 (see EcoStruxure™ Control Expert, System Bits and Words, Reference Manual) are raised except for the Modicon M340. The Modicon M340 PLCs are able to use the denormalized operands but, due to the format, with a loss of precision. Underflow is signaled depending on the operation only when the result is 0 (total underflow) or when the result is a denormalized (gradual underflow, with loss of precision).

The Real Type

Presentation:

| Type | Scale (bits) | Default value |
|------|--------------|---------------|
| REAL | 32 | 0.0 |

Range of values (grayed out parts):



When a calculation result is:

- between $-1.1754944e-38$ and $1.1754944e-38$, it is a DEN
- less than $-3.4028234e+38$, the symbol `-INF` (for -infinite) is displayed
- greater than $+3.4028234e+38$, the symbol `INF` (for +infinite) is displayed
- undefined (square root of a negative number), the symbol `NAN` is displayed

Examples of inaccuracy on normalized value

7.986 will be coded by the application as:

| S | E=129 | M=8359248 |
|---|---------|-------------------------|
| 0 | 1000001 | 11111111000110101010000 |

Using the formula:

$$(-1)^0 \times 2^{(129-127)} \times \left(1 + \frac{8359248}{2^{23}}\right) = (7,986000061035145625)$$

The number 7.986 should have a significant of:

$$\left(\frac{7,986}{2^2} - 1\right) \times 2^{23} = (8359247,872)$$

As the significant is expressed as an integer, it can only be coded as 8359248 (rounded to the nearest limit).

No number can be coded between the significant 8359247 and 8359248, or between the real number 7.985999584197998046875 and 7.98600006103515625

The weight of the less significant bit (gap) is, in absolute precision:

$$\frac{2^{(129-127)}}{2^{23}} = 2^{-21} = 0,000000476837158203125$$

The gap becomes very important for big values as shown below:

| Value | $Range \leq 2^{n-1} \leq Value \leq 2^n$ | M=8359248 |
|-------------|--|--|
| 100 000 000 | Between 2^{26} and 2^{27} | $\frac{2^{26}}{2^{23}} = 2^3 = 8$ |
| 2^{127} | 2^{127} | $\frac{2^{127}}{2^{23}} = 2^{104} = 2,02 \times 10^{31}$ |

NOTE: The gap corresponds to the weight of the less significant bit.

In order to get an expected resolution, it is necessary to define the maximum range for the calculation according the following formula:

$$e = \frac{\lceil \text{Ln}(p \times 2^{23}) \rceil}{\text{Ln}(2)}$$

p being the accuracy and e the exponent (e = E-127)

For instance, if the accuracy needs to be = 0.001, the fixed-point part will be:

$$F = (-1)^S \times 2^{(e)} \times \left(1 + \frac{M}{2^{23}}\right) = 2^{14} = 16384$$

with:

$$13 = \frac{\lceil \text{Ln}(0,001 \times 2^{23}) \rceil}{\text{Ln}(2)}$$

Beyond of this limit F, the accuracy will be lost.

Typical case: Counters

Floating must be used carefully, especially when it needs to add a small number to itself.

In case of small increments, the counter won't count properly, giving wrong results and stopping to rise when the increment will be lower than the less significant bit of the counter.

To get correct values, it is recommended to count on an double integer (UDINT) and multiply the result by the increment.

Example:

- Increment a value by 0.001 from 33000 to 1000000,
- Count from 33000000 to 1000000000 (value times 1000) with 1 as increment,
- Get the result multiplying the value by 0.001.

The accuracy F minimum per range will be:

| From...to... | F (minimum) |
|------------------|-------------|
| 3300...65536 | 0.004 |
| 65536...131072 | 0.008 |
| ... | ... |
| 524288...1000000 | 0.063 |

This counter can raise up to $4294967295 \times 0.001 = 4294967.5$ with a minimum accuracy of 0.5

NOTE: The real value here are the binary value encoded. It may differs from the display in an operator screen as rounding is done (4.294968e+006)

Elementary Data Types (EDT) in Character String Format

Subject of this Section

This section describes data types in character string format.

Overview of Data Types in Character String Format

Introduction

Data types in character string format belong to the EDT (Elementary data type) family, which includes **single** rather than derived data types (tables, structures, function blocks).

The Character String Type

The character string format is used to represent a string of ASCII characters, with each character being coded in an 8 bit format.

The characteristics of character string types are as follows:

- 16 characters by default in a string (excluding end of string characters)
- a string is composed of ASCII characters between 16#20 and 16#FF (hexadecimal representation)
- in an empty string, the end of string character (code ASCII "ZERO") is the first character of the string
- the maximum size of a string is 65535 characters

The size of the character string can be optimized during the definition of the type using the **STRING[<size>]** command, <size> being an unsigned integer UINT capable of defining a string of between 1 and 65535 ASCII characters.

NOTE: The ASCII characters 0-127 are common to all languages, but the characters 128-255 are language dependent. Be careful if the language of the Control Expert is not the same as the OS language. If the two languages are not the same, CHAR MODE communication can be disturbed and sending characters greater than 127 cannot be guaranteed to be correct. In particular, if the "Stop on Reception" character is greater than 127, it is not taken into account.

Syntax Rules

The entry is preceded by and ends with the quote character "" (ASCII code 16#27).

The \$ (dollar) sign is a special character, followed by certain letters which indicate:

- \$L or \$l, go to the next line (line feed)
- \$N or \$n, go to the start of the next line (new line)
- \$P or \$p, go to the next page
- \$R or \$r, carriage return
- \$T or \$t tabulation (Tab)
- \$\$, represents the character \$ in a string
- \$', represents the quote character in a string

The user can use the syntax \$nn to display, in a STRING variable, characters which must not be printed. It can be a carriage return (ASCII code 16#0D) for instance.

Examples

Entry examples:

| Type | Entry | Contents of the string • represents the end of string character * represents empty bytes |
|------------|--------------|--|
| STRING | 'ABCD' | ABCD•***** (16 characters) |
| STRING[4] | 'john' | john• |
| STRING[10] | 'It's john' | It's john•* |
| STRING[5] | '' | •***** |
| STRING[5] | '\$' | '•***** |
| STRING[5] | 'the number' | the no• |
| STRING[13] | '0123456789' | 0123456789•*** |
| STRING[5] | '\$R\$L' | <cr><lf>•*** |
| STRING[5] | '\$\$1.00' | \$1.00• |

STRING Type Variable Declaration

A STRING type variable can be declared in two different ways:

- STRING and
- STRING[<Number of elements>]

Behavior differs depending on usage:

| Type | Variable declaration | FFB input parameter | EF output parameter | FB output parameter |
|-------------|---------------------------|--|--|--|
| STRING | Fixed size: 16 characters | The size is equal to the actual size of the input parameter. | The size is equal to the actual size of the input parameter. | Fixed size of 16 characters |
| STRING[<n>] | Fixed size: n characters | The size is equal to the actual size of the input parameter limited to n characters. | The EF writes a maximum of n characters. | The FB writes a maximum of n characters. |

Strings and the ANY Pin

When you use a STRING type variable as an ANY type parameter, it is highly recommended to check that the size of the variable is less than the maximum declared size.

Example:

Use of STRING on the SEL function (Selector).

```
String1: STRING[8]
```

```
String2: STRING[4]
```

```
String3: STRING[4]
```

```
String1:= 'AAAAAAAA';
```

```
String3:= 'CC';
```

```
Scenario 1:
```

```
String2:= 'BBBB';
```

```
(* the size of the string is equal to the maximum declared size *)
```

```
String1:= SEL(FALSE, String2, String3);
```

```
(* the result will be: 'BBBBAAAA' *)
```

```
Scenario 2:
```

```
String2:= 'BBB';
```

```
(* the size of the string is less than the maximum declared size *)
```

```
String1:= SEL(FALSE, String2, String3);
```

```
(* the result will be: 'BBB' *)
```

Elementary Data Types (EDT) in Bit String Format

Subject of this Section

This section describes data types in bit string format. These are:

- Byte type
- Word type

- Dword type

Overview of Data Types in Bit String Format

Introduction

Data types in bit string format belong to the EDT (Elementary data type) family, which includes **single** rather than derived data types (tables, structure, function blocks).

Reminder Concerning Bit String Format

The particularity of this format is that all of its component bits do not represent a numerical value, but a combination of separate bits.

The data belonging to types of this format can be represented in three bases. These are:

- hexadecimal (16#)
- octal (8#)
- binary (2#)

Data Types in Bit String Format

Three data types:

| Type | Scale (bits) | Default value |
|-------|--------------|---------------|
| BYTE | 8 | 0 |
| WORD | 16 | 0 |
| DWORD | 32 | 0 |

Bit String Types

The Byte Type

The Byte type is coded in 8 bit format.

This table shows the lower/upper limits of the bases which can be used.

| Base | Lower limit | Upper limit |
|-------------|-------------|-------------|
| Hexadecimal | 16#0 | 16#FF |
| Octal | 8#0 | 8#377 |
| Binary | 2#0 | 2#11111111 |

Representation examples:

| Data content | Representation in one of the bases |
|--------------|------------------------------------|
| 00001000 | 16#8 |
| 00110011 | 8#63 |
| 00110011 | 2#110011 |

The Word Type

The Word type is coded in 16 bit format.

This table shows the lower/upper limits of the bases which can be used.

| Base | Lower limit | Upper limit |
|-------------|-------------|--------------------|
| Hexadecimal | 16#0 | 16#FFFF |
| Octal | 8#0 | 8#177777 |
| Binary | 2#0 | 2#1111111111111111 |

Representation examples:

| Data content | Representation in one of the bases |
|------------------|------------------------------------|
| 0000000011010011 | 16#D3 |
| 1010101010101010 | 8#125252 |
| 0000000011010011 | 2#11010011 |

the Dword Type

The Dword type is coded in 32 bit format.

This table shows the lower/upper limits of the bases which can be used.

| Base | Lower limit | Upper limit |
|-------------|-------------|------------------------------------|
| Hexadecimal | 16#0 | 16#FFFFFFFF |
| Octal | 8#0 | 8#3777777777 |
| Binary | 2#0 | 2#11111111111111111111111111111111 |

Representation examples:

| Data content | Representation in one of the bases |
|----------------------------------|------------------------------------|
| 00000000000010101101110011011110 | 16#ADCDE |
| 000000000000001000000000000000 | 8#200000 |
| 0000000000001010101110011011110 | 2#10101011110011011110 |

Derived Data Types (DDT/IODDT/Device DDT)

Subject of this Section

This section presents Derived Data Types. These are:

- tables (DDT)
- structures
 - structures concerning input/output data (IODDT)
 - structures concerning other data (DDT)
 - structures concerning input/output data.

Arrays

What Is an Array?

It is a data item that contains a **set of data of the same type**, such as:

- elementary data (EDT),
for example:
 - a group of BOOL words,
 - a group of UINT integer words,
 - etc.
- derived data (DDT),
for example:

- a group of WORD tables,
- a group of structures,
- Device derived data (Device DDT)
- etc.

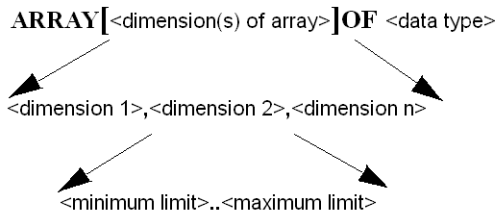
Characteristics

An array is characterized by two parameters:

- a parameter which defines its organization (array dimension(s)),
- a parameter that defines the type of data it contains.

NOTE: The most complex organization is the array with **15 dimensions** and the array size could not be greater than 65535 bytes.

The syntax comprising these two parameters is:



`<minimum limit>` **strictly less** than `<maximum limit>`

Defining and Instancing an Array

Definition of an array type:

```
X: ARRAY[1..10] OF BOOL
```

Instancing an array

```
Tab_1: X
Tab_2: ARRAY[1..10] OF BOOL
```

The instances Tab_1 and Tab_2 are of the same type and the same dimension, the only difference being that during instancing:

- the Tab_1 type takes the name X,
- the Tab_2 type must be defined (unnamed table).

NOTE: It is beneficial to name the type, as any modification that has to be made will only be done so once, otherwise there will be as many modifications as there are instances.

Examples

This table presents the instances of arrays of different dimensions:

| Entry | Comments |
|---|--|
| Tab_1: ARRAY[1..2] OF BOOL | 1 dimensional array with 2 Boolean words |
| Tab_2: ARRAY[-10..20] OF WORD | 1 dimensional array with 31 WORD type structures (structure defined by the user) |
| Tab_3: ARRAY[1..10, 1..20] OF INT | 2 dimensional arrays with 10x20 integers |
| Tab_4: ARRAY[0..2, -1..1, 201..300, 0..1] OF REAL | 4 dimensional arrays with 3x3x100x2 reals |

NOTE: Many functions (READ_VAR, WRITE_VAR for example) don't recognize the index of an array of words starting by a number different from 0. If you use such an index the functions will look at the number of words in the array, but not at the starting index set in the definition of the array.

| |
|---|
| ⚠ WARNING |
| UNEXPECTED APPLICATION BEHAVIOR - INVALID ARRAY INDEX |
| When applying functions on variables of array type, check that the functions are compatible with the arrays starting index value when this value is greater than 0. |
| Failure to follow these instructions can result in death, serious injury, or equipment damage. |

Access to a data item in array Tab_1 and Tab_3:

```
Tab_1[2]
;To access second element
```

```
Tab_3[4][18]
;To access eighteenth element of the fourth sub-array
```

Inter-Arrays Assignment Rules

There are the 4 following arrays:

```
Tab_1:ARRAY[1..10] OF INT
Tab_2:ARRAY[1..10] OF INT
Tab_3:ARRAY[1..11] OF INT
Tab_4:ARRAY[101..110] OF INT
```

```
Tab_1:=Tab_2; Assignment authorized
Tab_1:=Tab_3; Assignment refused (non-IEC compliant)
Tab_1:=Tab_4; Assignment refused (non-IEC compliant)
```

Structures

What is a Structure?

It is a data item containing a **set** of data **of a different type**, such as:

- a group of BOOL, WORD, UNINT, etc. , (EDT structure),
- a group of tables (DDT structure),
- a group of REAL, DWORD, tables, etc., (EDT and DDT structures).

NOTE: You can create nested structures (nested DDTs) over 15 levels. **Recurring structures (DDT) are not allowed.**

Characteristics

A structure is composed of data which are each characterized by:

- a type,
- a name, which enables it to be identified,
- a comment (optional) describing its role.

Definition of a structure type:

```
IDENT
  Surname: STRING[12]
  First name: STRING[16]
  Age: UINT

;The IDENT type structure contains a UINT type data item and two
STRING type data
```

Definition of two data instances of an IDENT type structure:

```
Person_1: IDENT
Person_2: IDENT

;The instances Person_1 and Person_2 are of IDENT Structure type
```

Access to the Data of a Structure

Access to the data of the Person_1 IDENT-type instance:

```
Person_1.Name ;To access name of Person_1
Person_1.Age  ;To access age of Person_1
```

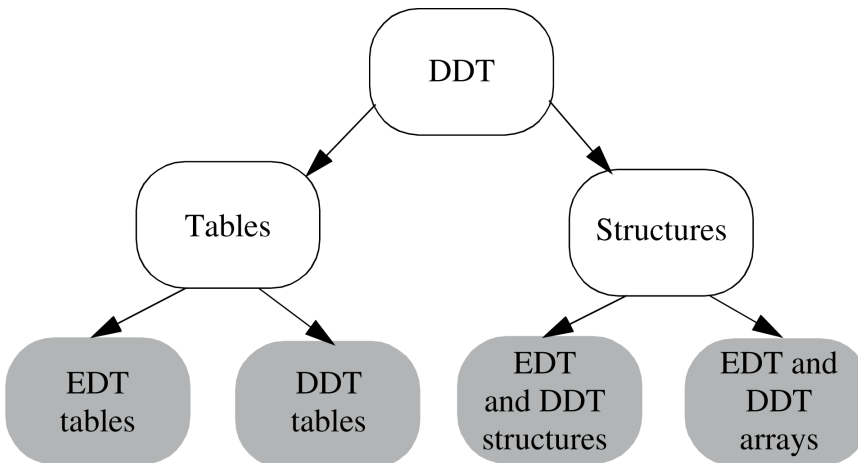
Overview of the Derived Data Type family (DDT)

Introduction

The DDT (Derived Data Type) family includes **"derived"** data types such as:

- tables
- structures

Illustration:



Characteristics

A data item belonging to the DDT family is made up of:

- the type name, page 164 (32 characters maximum) defined by the user (not obligatory for tables but recommended), page 193
- the type (structure or table)
- an optional comment (of a maximum of 1024 characters). Authorized characters correspond to the ASCII codes 32 to 255
- the description (in the case of a structure) of these elements
 - the element name, page 164 (32 characters maximum)
 - the element type
 - an optional comment (1024 characters maximum) describing its role. The authorized characters correspond to the ASCII codes 32 to 255
- information such as:
 - type version number
 - date of the last modification of the code or of the internal variables or of the interface variables
 - an optional descriptive file (32767 characters) describing the block function and its different modifications

NOTE:

1. M580 Platform and PLC Simulator: the total size of a table does not exceed 2 Mbytes for DDT structures and DDT arrays
2. Other Platforms: the total size of a table does not exceed 64 Kbytes

Examples

Definition of types

```
COORD
  X: INT
  Y: INT
;COORD type structure

SEGMENT
  Origin: COORD
  Destination: COORD
;SEGMENT type structure containing 2 COORD type
structures

OUTLINE: ARRAY[0..99] OF SEGMENT
;OUTLINE type table containing 100 SEGMENT type
structures

DRAW
  Color: INT
  Anchor: COORD
  Pattern: ARRAY[0..15,0..15] OF WORD
  Contour: OUTLINE
```

Access to the data of a DRAW-type structure instance

```
Cartoon: DRAW
;Instance of DRAW type structure

Cartoon.Pattern[15,15]
;Access to last data item in the Pattern table of the
Cartoon structure

Cartoon.Contour[0].Origin.X
;Access to data item X of the COORD structure belonging to
the first SEGMENT structure of the Contour table.
```

DDT: Mapping Rules

At a Glance

The DDTs are stored in the memory of the PLC in the order in which its elements are declared.

However, the following rules apply.

Principle for Premium and Quantum

The storage principle for Premium and Quantum is as follows:

- the elements are stored in the order in which they are declared in the structure,
- the basic element is the byte (alignment of data on the memory bytes),
- each element has an alignment rule:
 - the `BOOL` and `BYTE` types are indiscriminately aligned on the odd or even bytes,
 - all the other elementary types are aligned on the even bytes,
 - the structures and tables are aligned according to the alignment rule for the `BOOL` and `BYTE` types if they only contain `BOOL` and `BYTE` elements, otherwise they are aligned on the memory's even bytes.

WARNING

RISK OF INCOMPATIBILITY AFTER CONCEPT CONVERSION

With the **Concept** programming application, the data structures do not handle any shift in offsets (each element is set one after the other in the memory, regardless of its type). Consequently, we recommend that you check everything, in particular the consistency of the data when using DDTs located on the "State RAM" (risk of shifts) or functions for communication with other devices (transfers with a different size from those programmed in Concept).

Failure to follow these instructions can result in death, serious injury, or equipment damage.

Principle for Modicon M340, M580, and Momentum

The storage principle for these PLCs is as follows:

- elements are stored in the order in which they are declared in the structure,
- the basic element is the byte,
- one alignment rule and function of the element:

- the `BOOL` and `BYTE` types are aligned on either even or uneven bytes,
- the `INT`, `WORD` and `UINT` types are aligned on even bytes,
- the `DINT`, `UDINT`, `REAL`, `TIME`, `DATE`, `TOD`, `DT` and `DWORD` are aligned on double words,
- structures and tables are aligned according to the rules of their elements.

⚠ WARNING

BAD EXCHANGES BETWEEN A MODICON M340, M580, MOMENTUM AND A PREMIUM OR QUANTUM.

Check if the structure of the exchanged data have the same alignments in the two projects. Otherwise, the data will not be exchanged properly.

Failure to follow these instructions can result in death, serious injury, or equipment damage.

NOTE: It is possible that the alignment of data are not the same when the project is transferred from the simulator of Control Expert to a M340, M580, or Momentum PLC. So check the structure of the data of the project.

NOTE: Control Expert indicates where the alignment seems to be different. Check the corresponding instances in the data editor. See the page of Project settings (see EcoStruxure™ Control Expert, Operating Modes) to know how enable this option.

Modicon M580 Device DDT Alignment for I/O Scanning

Two modes of I/O scanning are proposed:

- Legacy I/O scanning mode (used in Unity Pro \leq V11.1) creates Device DDT structures aligned on 32 bits by default.
- Enhanced I/O scanning mode (compatible for applications created with Unity Pro \geq V12.0) creates Device DDT structures aligned on 16 bits by default.

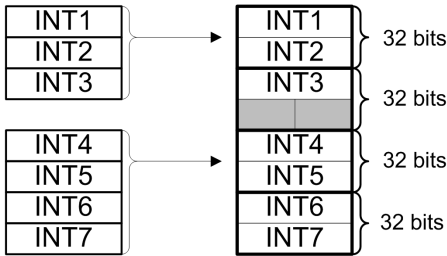
NOTE:

Unity Pro is the former name of Control Expert for version 13.1 or earlier.

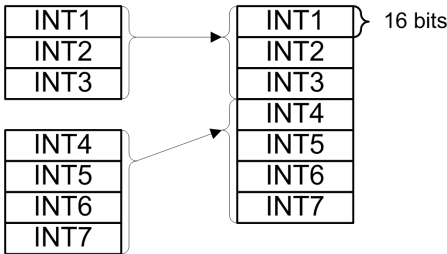
To keep the original alignment for applications created for Unity Pro \leq V11.1, select the legacy I/O scanning mode.

Alignment mismatch illustration for Modbus TCP device:

- Example of arrangement in legacy mode (32 bits alignment, array of 4 x `BYTE`). When 3 `INT` are transmitted, the structure in legacy mode creates 2 empty bytes that need to be considered in the global structure interpretation.



- Example of arrangement in enhanced mode (16 bits alignment, array of `INT`). When 3 `INT` are transmitted, no empty bytes are added by the system, all the data in the structure are useful.



Examples

The following table gives some examples of data structures. In the following examples, structure type DDTs are addressed to `%MWi`. The word's first byte corresponds to the least significant 8 bits and the word's second byte corresponds to the most significant 8 bits.

For all the following structures, the first variable is mapped to the address `%MW100`:

| First Memory Address | | Description of the structure |
|-----------------------------------|-----------------------------------|------------------------------|
| Modicon M340, M580 or Momentum | Premium | Para_PWM1 |
| <code>%MW100</code> (first byte) | <code>%MW100</code> (first byte) | t_period: TIME |
| <code>%MW102</code> (first byte) | <code>%MW102</code> (first byte) | t_min: TIME |
| <code>%MW104</code> (first byte) | <code>%MW104</code> (first byte) | in_max: REAL |
| | | |
| | | Mode_TOTALIZER |
| <code>%MW100</code> (first byte) | <code>%MW100</code> (first byte) | hold: BOOL |
| <code>%MW100</code> (second byte) | <code>%MW100</code> (second byte) | rst: BOOL |
| | | |

| First Memory Address | | Description of the structure | |
|----------------------|----------------------|------------------------------|-----------------------------------|
| | | Info_TOTALIZER | |
| %MW100 (first byte) | %MW100 (first byte) | | outc: REAL |
| %MW102 (first byte) | %MW102 (first byte) | | cter: UINT |
| %MW103 (first byte) | %MW103 (first byte) | | done: BOOL |
| %MW103 (second byte) | %MW103 (second byte) | | Reserved for the alignment |

The table below gives two examples of data structures with arrays:

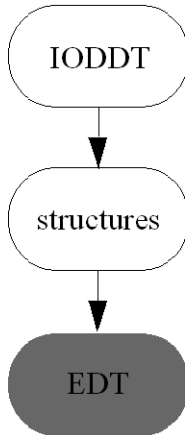
| First Memory Address | | Description of the structure | |
|--------------------------------|----------------------|------------------------------|---|
| Modicon M340, M580 or Momentum | Premium | EHC105_Out | |
| %MW100 (first byte) | %MW100 (first byte) | | Quit: BYTE |
| %MW100 (second byte) | %MW100 (second byte) | | Control: ARRAY [1..5] OF BYTE |
| %MW104 (first byte) | %MW103 (first byte) | | Final: ARRAY [1..5] OF DINT |
| | | | |
| | | CPCfg_ex | |
| %MW100 (first byte) | %MW100 (first byte) | | Profile_type: INT |
| %MW101 (first byte) | %MW101 (first byte) | | Interp_type: INT |
| %MW102 (first byte) | %MW102 (first byte) | | Nb_of_coords: INT |
| %MW103 (first byte) | %MW103 (first byte) | | Nb_of_points: INT |
| %MW104 (first byte) | %MW104 (first byte) | | reserved: ARRAY [0..4] OF BYTE |
| %MW106 (second byte) | %MW106 (second byte) | | Reserved for the alignment of variable Master_offset on even bytes |
| %MW108 (first byte) | %MW107 (first byte) | | Master_offset: DINT |
| %MW110 (first byte) | %MW109 (first byte) | | Follower_offset: INT |
| %MW111 (entire word) | - | | Reserved for the alignment |

Overview of Input/Output Derived Data Types (IODDT)

At a Glance

The IODDTs (Input Output Derived Data Types) **are predefined by the manufacturer**, and contain language objects of the EDT family belonging to the channel of an application-specific module.

Illustration:



The IODDT types are structures whose size (the number of elements of which they are composed) depends on the channel or the input/output module that they represent.

A given input/output module can have more than one IODDT.

The difference with a conventional structure is that:

- the IODDT structure is predefined by the manufacturer
- The elements comprising the IODDT structure do not have a contiguous memory allocation, but rather a specific address in the module

Examples

IODDT structure for an input\output channel of an analog module

```
ANA_IN_GEN      ;ANA_IN_GEN type structure
  Value:INT      ;Input value
  Err:  BOOL     ;Channel error
```

Access to the data of an instance of the ANA_IN_GEN type:

```
Cistern_Level: ANA_IN_GEN
; ANA_IN_GEN type instance which corresponds for example
to a tank level sensor
```

```
Cistern_Level.Value ;Reading of the channel input value
Cistern_Level.Err   ;Reading of channel error bit
```

Access by direct addressing:

For channel 0 of module 2 of rack 0 we obtain:

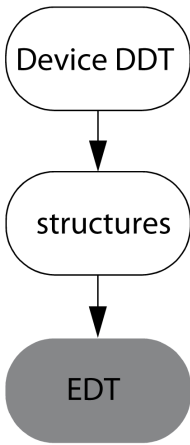
```
Cistern_Level      corresponds to %CH0.2.0
Cistern_Level.Value corresponds to %IW0.2.0.0
Cistern_Level_Err  corresponds to %IO.2.0.ERR
```

Overview of Device Derived Data Types (Device DDT)

At a Glance

A Device DDT is a DDT predefined by the manufacturer and not modifiable by user. It contains the I/O language elements of an I/O Module.

Illustration:



Device DDT structures like DDT structures can contain:

- EDT
- DDT
- Array of EDT and DDT

The DDT types are structures whose size (the number of elements of which they are composed) depends on the channel or the input/output module that they represent.

In the current implementation, an I/O Module supports only one Device DDT type.

The difference with a conventional structure is that:

- the DDT structure is predefined by the manufacturer
 - the DDT structure supports EBOOL
 - the DDT structure supports type with extracted bits

Device DDT Instance Naming Rule

Modules Concerned by Default Naming Rule

The following table presents the main categories of modules concerned by the device DDT instance naming rule:

| Architecture | | Modicon M580 | Modicon Quantum |
|--------------|--------|----------------------------|-----------------|
| Position | Family | | |
| Local Drop | X80 | Analog I/O: most modules | - |
| | | Discrete I/O: most modules | |

| Architecture | | Modicon M580 | Modicon Quantum |
|--------------------------|------------------------|---|---|
| Position | Family | | |
| | | Counting: most modules | |
| | | Communication: <ul style="list-style-type: none"> • BMXEIA0100 • BMXNOM0200 • BMECXM0100 and slaves attached | |
| | Modicon Quantum | – | |
| Ethernet I/O Drop | X80 | BM•CRA312•• adapter module | BM•CRA312•• adapter module |
| | | Analog I/O: most modules | Analog I/O: most modules |
| | | Discrete I/O: most modules | Discrete I/O: most modules |
| | | Counting: most modules | Counting: most modules |
| | | Communication: <ul style="list-style-type: none"> • BMXEIA0100 • BMXNOM0200 • BMECXM0100 and slaves attached | Communication: <ul style="list-style-type: none"> • BMXEIA0100 • BMXNOM0200 • BMECXM0100 and slaves attached |
| | Modicon Quantum | The drop | The drop |
| | | Analog I/O: most modules | – |
| | | Discrete I/O: most modules | |
| | | Counting: No | |
| | | Communication: No | |

Default Naming Rule

The syntax is based on topological naming and is built as follows:

BBBx_dx_rx_sx_PPPPPPP_SSS

- *BBBx*: Bus name and bus number.
 - *BBB* = Bus name represented by the 3 first characters of the bus name displayed in the Control Expert project browser.
 - *x* = Bus number
- *dx*: Drop number.
 - *d* = *d*

- x = Drop number. Number equals 0 for a virtual drop.
- rx : Rack number.
 - $r = r$
 - x = Rack number. Number equals 0 for a virtual rack, optional for CANopen devices.
- sx : Slot number.
 - $s = s$
 - x = Slot number. Optional for CANopen devices.
- $PPPPPP$: Device part number. Part number without space as it is displayed on the device representation in Control Expert.
- SSS : Name of a subset if the device DDT is linked to a subset. These characters are optional.

NOTE: If a name is not unique, $_rrrrr$ is added at the end of the string ($rrrrr$ being a random character series).

Examples

Device DDT instance name examples in a Modicon M580 application (M580 CPU):

- Modicon M580 local drop 0, rack 0, slot 2 located on **PLC bus** number 0. BMXDAI0805 module.

PLC0_d0_r0_s2_DAI0805

- X80 Ethernet I/O drop 1, rack 0, slot 0 located on **EIO bus** number 2. BMXCRA31200 module.

EIO2_d1_r0_s0_CRA31200

- Modicon Quantum Ethernet I/O drop 2, rack 1 located on **EIO bus** number 2. Modicon Quantum drop with a 140CRA31200 adapter module.

EIO2_d2_DROP

NOTE: In this case, the rack and slot numbers are omitted.

Device DDT instance name examples in a Modicon Quantum application (Quantum CPU):

- Modicon Quantum local drop 1, rack 1, slot 4 located on **Local Bus** number 2. 140CRP31200 adapter module to address Ethernet I/O drops.

LOC1_d1_r1_s4_CRP31200

- X80 Ethernet I/O drop 1, rack 0, slot 0 located on **EIO bus** number 2. BMECRA31210 module.

EIO2_d1_ECRA31210

NOTE: In this case, the rack and slot numbers are omitted as for a Quantum Ethernet I/O drop adapter.

- X80 Ethernet I/O drop 1, rack 0, slot 1 located on **EIO bus** number 2. BMXDAI0805 module.
EIO2_d1_r0_s1_DAI0805
- Modicon Quantum Ethernet I/O drop 2, rack 1 located on **EIO bus** number 2. Modicon Quantum drop with a 140CRA31200 adapter module.

EIO2_d2_DROP

NOTE: In this case, the rack and slot numbers are omitted.

Renaming a Device DDT, Copying, Pasting and Moving a Module

Actions on Device DDT instances and modules with an associated device DDT are detailed in the following section: *Managing a Device DDT instance* (see EcoStruxure™ Control Expert, Operating Modes).

Function Block Data Types (DFB\EFB)

Subject of this Section

This section describes function block data types. These are:

- user function blocks (DFB)
- elementary function blocks (EFB)

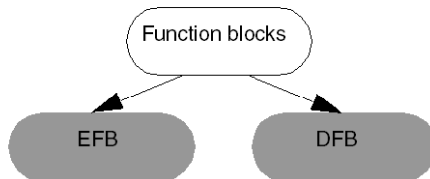
Overview of Function Block Data Type Families

Introduction

Function block data type families are:

- the Elementary Function Block (EFB), page 160 type family
- the User function block (DFB), page 160 type family

Illustration:

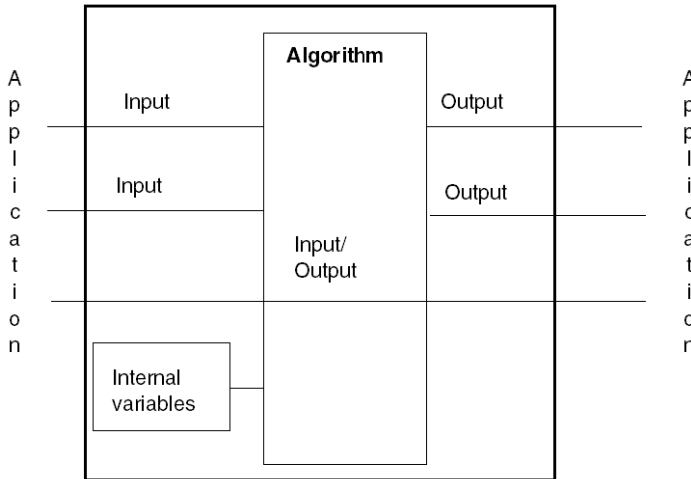


Function blocks are entities containing:

- input and output variables acting as an interface with the application
- a processing algorithm that operates input variables and completes the output variables
- private and public internal variables operated by the processing algorithm

Illustration

Function block:



User Function Block (DFB)

The user function block types (Derived Function Blocks) are developed by the user using one or more languages (according to the number of sections). These languages are:

- Ladder language
- Structured Text language
- Instruction List language
- Functional block language FBD

A DFB type can have one or more instances where each instance is referenced by a name (symbol) and possesses DFB data types.

Elementary Function Block (EFB)

Elementary Function Blocks (EFBs) are provided by the manufacturer and are programmed in C language.

The user can create his own EFB for which he will need an optional software tool "**SDKC**".

An EFB type can have one or more instances where each instance is referenced by a name (symbol) and possesses EFB type data.

Characteristics of Function Block Data Types (EFB\DFB)

Type Definition

The type of an EFB or DFB function block is defined by:

- the type name, page 164, defined by the user for the DFBs,
- an optional comment. The authorized characters correspond to the ASCII codes 32 to 255,
- the application interface data:
 - the inputs, not accessible in read\write mode from the application, but read by the function block code,
 - the inputs\outputs, not accessible in read\write mode from the application, but read and written by the function block code,
 - the outputs, accessible in read only from the application and read and written by the function block code.
- the internal data:
 - public internal data, accessible in read\write mode from the application, and read and written by the function block code,
 - private internal data, not accessible from the application, but read and written by the function block code.
- the code:
 - for DFBs, this is written by the user in PLC language (Structured Text, Instruction List, Ladder language, function block language), and is structured in a single section or in several sections,
 - for EFBs, this is written in C language.
- information such as:
 - type version number,
 - date of the last modification of the code, or of the internal variables, or of the interface variables.

- an optional descriptive file (32767 characters), describing the block function and its different modifications.

Characteristics

This table gives the characteristics of the elements that make up a type:

| Element | EFB | DFB |
|---|-----------------------|--|
| Name | 32 characters | 32 characters |
| Comment | 1024 characters | 1024 characters |
| Input Data | 32 maximum | 32 maximum |
| Input/Output data | 32 maximum | 32 maximum |
| Output data | 32 maximum | 32 maximum |
| Number of interfaces (Inputs+Outputs+Inputs/Outputs) | 32 maximum (2) | 32 maximum (2) |
| Public data | No limits (1) | No limits (1) |
| Private data | No limits (1) | No limits (1) |
| Programming language | C language | Language: <ul style="list-style-type: none"> • Structured Text, • Instruction List, • Ladder language, • function block. |
| Section | | <p>A section is defined by:</p> <ul style="list-style-type: none"> • a name (maximum 32 characters), • a validation condition, • a comment (maximum 256 characters), • a protection: <ul style="list-style-type: none"> ◦ without, ◦ read only, ◦ read\write mode. <p>A section cannot access declared variables in the application, except for:</p> <ul style="list-style-type: none"> • system double words %SDi, • system words %SWi, • system bits %Si. |

(1): the only limit is the size of the PLC's memory.

(2): the EN input and ENO output are not taken into account.

Characteristics of Elements Belonging to Function Blocks

What is an element?

Each element (interface data or internal data) is defined by:

- a name, page 164 (maximum 32 characters), defined by the user,
- a type,
 - which can belong to the following families:
 - Elementary Data Types (EDT),
 - Derived Data Type (DDT),
 - Device Derived Data Type (Device DDT)
 - Function Block data types (EFB\DFB).
- an optional comment (maximum 1024 characters). The authorized characters correspond to the ASCII codes 32 to 255,
- an initial value,
- an access right from the application program (sections of the application or section belonging to the DFBs see "Definition of the function block type (interface and internal variables)", page 210,
- an access right from communication requests,
- a public variables backup flag.

Authorized Data Types for an Element Belonging to a DFB

The authorized data types are:

| Element of the DFB | EDT types | DDT types | | | | ANY... | Function block types |
|--------------------|-----------|-----------|----------------|-----------|-------|-------------|----------------------|
| | | IODDT | Unnamed tables | ANY_ARRAY | other | | |
| Input data | Yes | No | Yes | Yes | Yes | Yes (2) | No |
| Input/output data | Yes (1) | Yes | Yes | Yes | Yes | Yes (2) | No |
| Output data | Yes | No | Yes | No | Yes | Yes (2) (3) | No |
| Public data | Yes | No | Yes | No | Yes | No | No |
| Private data | Yes | No | Yes | No | Yes | No | Yes |

- (1): not authorized for the EBOOL type static data used on Quantum PLCs
- (2): not authorized for BOOL and EBOOL type data
- (3): must be completed during the execution of the DFB, and not usable outside the DFB

Authorized Data Types for an Element Belonging to an EFB

The authorized data types are:

| Element of the EFB | EDT types | DDT types | | | | ANY... | Function block types |
|--------------------|-----------|-----------|----------------|------------|-------|----------------|----------------------|
| | | IODDT | Unnamed tables | ANY_AR-RAY | other | | |
| Input data | Yes | No | No | Yes | Yes | Yes (1) | No |
| Input/output data | Yes | Yes | No | Yes | Yes | Yes (1) | No |
| Output data | Yes | No | No | No | Yes | Yes (1) (2) | No |
| Public data | Yes | No | No | No | Yes | No | No |
| Private data | Yes | No | No | No | Yes | No | Yes |

- (1): not authorized for BOOL and EBOOL type data
- (2): must be completed during the execution of the EFB, and not usable outside the EFB

Initial Values for an Element Belonging to a DFB

This table specifies whether the initial values can be entered from the DFB type definition or the DFB instance:

| Element of the DFB | From the DFB type | From the DFB instance |
|------------------------------|-------------------|-----------------------|
| Input data (no ANY... type) | Yes | Yes |
| Input data (of ANY... type) | No | No |
| Input/output data | No | No |
| Output data (no ANY... type) | Yes | Yes |
| Output data (of ANY... type) | No | No |
| Public data | Yes | Yes |
| Private data | Yes | No |

Initial Values for an Element Belonging to an EFB

This table specifies whether the initial values can be entered from the EFB type definition or the EFB instance:

| Element of the EFB | From the EFB type | From the DFB instance |
|--|-------------------|-----------------------|
| Input data (no ANY... type See generic data types, page 214) | Yes | Yes |
| Input data (of ANY... type) | No | No |
| Input/output data | No | No |
| Output data (no ANY... type) | Yes | Yes |
| Output data (of ANY... type) | No | No |
| Public data | Yes | Yes |
| Private data | Yes | No |

⚠ WARNING

UNEXPECTED APPLICATION BEHAVIOR - INVALID ARRAY INDEX

When using EFBs and DFBs on variables of array type, only use arrays with starting index=0.

Failure to follow these instructions can result in death, serious injury, or equipment damage.

Generic Data Types (GDT)

Subject of this Section

This section describes Generic Data Types (GDT).

Overview of Generic Data Types

At a Glance

Generic Data Types are conventional groups of data types (EDT, DDT) specifically intended to determine compatibility among these conventional groups of data types.

These groups are identified by the prefix 'ANY_ARRAY', but these prefixes can under no circumstances be used to instance the data.

Their field of use concerns function block (EFB\DFB) and elementary function (EF) data type families, in order to define which data types are compatible with their interfaces for the following :

- inputs
- input/outputs
- outputs

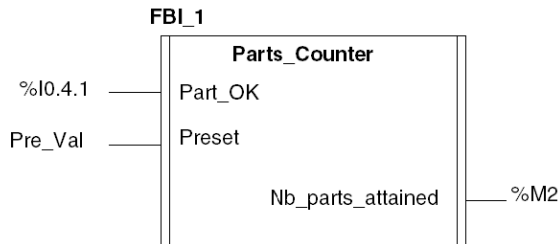
Available Generic Data Types

The generic data types available in Control Expert are the following types:

- ANY_ARRAY_WORD
- ANY_ARRAY_UINT
- ANY_ARRAY_UDINT
- ANY_ARRAY_TOD
- ANY_ARRAY_TIME
- ANY_ARRAY_STRING
- ANY_ARRAY_REAL
- ANY_ARRAY_INT
- ANY_ARRAY_EBOOL
- ANY_ARRAY_DWORD
- ANY_ARRAY_DT
- ANY_ARRAY_DINT
- ANY_ARRAY_DATE
- ANY_ARRAY_BYTE
- ANY_ARRAY_BOOL

Example

This gives us the following DFB:



The **Preset** input parameter may be defined of GDT-type.

NOTE: The authorized objects for the various parameters are defined in this table, page 470.

Data Types Belonging to Sequential Function Charts (SFC)

Subject of this Section

This section describes Sequential Function Chart (SFC) data types, which are structures predefined by the manufacturer.

Overview of the Data Types of the Sequential Function Chart Family

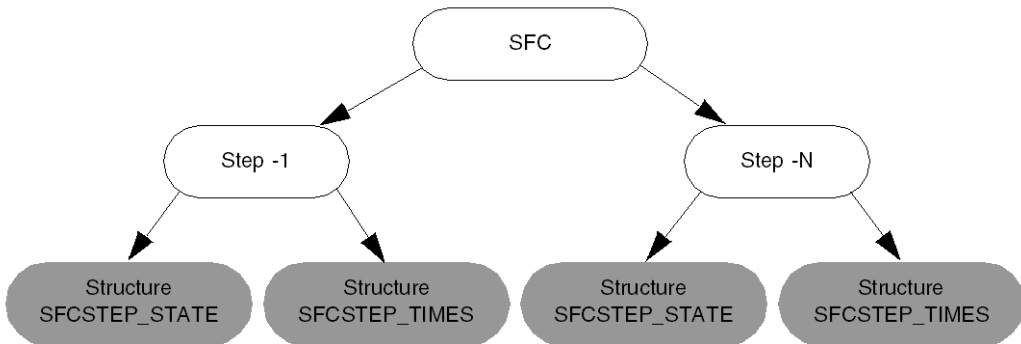
Introduction

The Sequential Function Chart (SFC) data type family includes **derived** data types, such as the structures that restore the properties and status of the chart and its component actions.

Each step is represented by two structures. These are:

- the **SFCSTEP_STATE** structure
- the **SFCSTEP_TIMES** structure

Illustration:



NOTE: The two structure types **SFCSTEP_STATE** and **SFCSTEP_TIMES** are also linked to each Macro step of the sequential function chart.

Definition of the SFCSTEP_STATE Structure Type

This structure includes all types of data linked to the status of the step or of the Macro step.

These data types are:

- **x**: BOOL elementary data type (EDT) containing the value TRUE when the step is active,
- **t**: TIME elementary data type (EDT) containing the activity time of the step. When deactivated, the step value is maintained until the next activation,
- **tminErr**: BOOL elementary data type (EDT) containing the value TRUE if the activity time of the step is less than the minimum programmed activity time,
- **tmaxErr**: BOOL elementary data type (EDT) containing the value TRUE if the activity time of the step is greater than the maximum programmed activity time,

These data types are accessible from the application in read only mode.

Definition of the SFCSTEP_TIMES Structure Type

This structure includes all types of data linked to the definition of the runtime parameters of the step or of the Macro step.

These data types are:

- **delay**: TIME elementary data type (EDT), defining the polling delay time of the transition situated downstream from the active step,

- **tmin**: TIME elementary data type (EDT) containing the minimum value during which the step must at least be executed. If this value is not respected the data tmin.Err switches to the value TRUE,
- **tmax**: TIME elementary data type (EDT) containing the maximum value during which the step must at least be executed. If this value is not respected the data tmax.Err switches to the value TRUE.

These data types are only accessible from the SFC editor.

Data Access Syntax of the Structure SFCSTEP_STATE

The instance names of this structure correspond to the names of the steps or macro steps of the sequential function chart

| Syntax | Comment |
|-------------------|---|
| Name_Step.x | Used to find out the status of the step (active/inactive) |
| Name_Step.t | Used to find out the current or total activation time for the step |
| Name_Step.tminErr | Used to find out if the minimum activation time of the step is less than the time programmed in Name_Step.tmin |
| Name_Step.tmaxErr | Used to find out if the maximum activation time of the step is greater than the time programmed in Name_Step.tmax |

Compatibility Between Data Types

Subject of this Section

This section presents compatibility among the following families of data types:

- the Elementary Data Type (EDT) family
- the Derived Data Type (DDT) family
- the Generic Data Type (GDT) family

Compatibility Between Data Types

Introduction

The following is a presentation of the different rules of compatibility between types **within** each of the following families:

- the Elementary Data Type (EDT) family

- the Derived Data Type (DDT) family
- the Generic Data Type (GDT) family

The Elementary Data Type (EDT) Family

The Elementary Data Type (EDT) family contains the following sub-families:

- the binary format data type sub-family
- the BCD format data type sub-family
- the Real format data type sub-family
- the character string format data type sub-family
- the bit string format data type sub-family

There is no compatibility whatsoever between two data types, even if they belong to the same sub-family.

Derived Data Type (DDT) Family

The Derived Data Type (DDT) family contains the following sub-families:

- the table type sub-family
- the structure type sub-family:
 - structures concerning input/output data (IODDT)
 - structures concerning input/output device (Device DDT)
 - structures concerning other data

Rules concerning the structures:

Two structures are compatible if their elements are:

- of the same name
- of the same type
- organized in the same order

There are four types of structure:

```
ELEMENT_1
  My_Element: INT
  Other_Element: BOOL
;ELEMENT_1 type structure
```

```
ELEMENT_2
  My_Element: INT
  Other_Element: BOOL
;ELEMENT_2 type structure
```

```
ELEMENT_3
  Element: INT
  Other_Element: BOOL
;ELEMENT_3 type structure
```

```
ELEMENT_4
  Other_Element: BOOL
  My_Element: INT
;ELEMENT_4 type structure
```

Compatibility between the structure types

| Types | ELEMENT_1 | ELEMENT_2 | ELEMENT_3 | ELEMENT_4 |
|-----------|-----------|-----------|-----------|-----------|
| ELEMENT_1 | | YES | NO | NO |
| ELEMENT_2 | YES | | NO | NO |
| ELEMENT_3 | NO | NO | | NO |
| ELEMENT_4 | NO | NO | NO | |

Rules concerning the tables

Two tables are compatible if:

- their dimensions and the order of their dimensions are identical
- each corresponding dimension is of the same type

There are five types of table:

```
TAB_1: ARRAY[10..20]OF INT
;Table one dimension of TAB_1 type

TAB_2: ARRAY[20..30]OF INT
;Table one dimension of TAB_2 type

TAB_3: ARRAY[20..30]OF INT
;Table one dimension of TAB_3 type

TAB_4: ARRAY[20..30]OF TAB_1
;Table one dimension of TAB_4 type

TAB_5: ARRAY[20..30,10..20]OF INT
;Table two dimensions of type TAB_5
```

Compatibility between the table types:

| Type... | and type... | are... |
|-----------|-------------|--------------|
| TAB_1 | TAB_2 | incompatible |
| TAB_2 | TAB_3 | compatible |
| TAB_4 | TAB_5 | compatible |
| TAB_4[25] | TAB_5[28] | compatible |

The Generic Data Type (GDT) Family

The Generic Data Type (GDT) family is made up of groups organized hierarchically which contain data types belonging to the following families:

- Elementary Data Types (EDT)
- Derived Data Types (DDT)

Rules:

A conventional data type is compatible with the genetic data types related to it hierarchically.

A generic data type is compatible with the generic data types related to it hierarchically.

Example:

The `INT` type is compatible with the `ANY_INT` or `ANY_NUM` or `ANY_MAGNITUDE` types.

The `INT` type is not compatible with the `ANY_BIT` or `ANY_REAL` types.

The `ANY_INT` generic type is compatible with the `ANY_NUM` type.

The `ANY_INT` generic type is not compatible with the `ANY_REAL` type.

Reference Data Type Declarations

At a Glance

This section explains the Reference data type.

Reference Data Type Declarations

Introduction

The Reference data type allows mapping of different types of data in a DDT.

A reference contains the memory address of a variable.

NOTICE

UNEXPECTED APPLICATION BEHAVIOR

Take specific care during your application testing to verify correct usage of references in your program.

Failure to follow these instructions can result in equipment damage.

References are declared using the keyword **REF_TO** followed by the type of the referenced value (for example: `myRefInt: REF_TO INT`).

A reference can be assigned to another reference if it points to the same or compatible data type (for example, `myRefINT1 := myRefINT2`).

References can be assigned to parameters of functions.

Summary of Control Expert reference operations:

| Operation | Description | Example |
|----------------------|--|--|
| Declaration | Declaration of a variable to be a reference | <code>myRefInt of type REF_TO INT</code> |
| Assignment | Assigns reference to reference (same type) | <code>myRefINT1 := myRefINT2;</code> |
| | Assigns reference to parameter of a function | <code>myFB (r := myRef);</code> |
| Comparison with NULL | | <code>IF myRef = NULL THEN ...</code> |
| Referencing | Assigns address of a variable to a reference | <code>myRefA := REF (A);</code> |
| Dereferencing | Provides the value of the variable referenced to | <code>A := myRefA^;</code> <code>B := myRefArrayType^[12];</code> |

A reference can be dereferenced using a postfix “^” (caret), but dereferencing a NULL reference produces a detected error.

Reference Limitations

A reference:

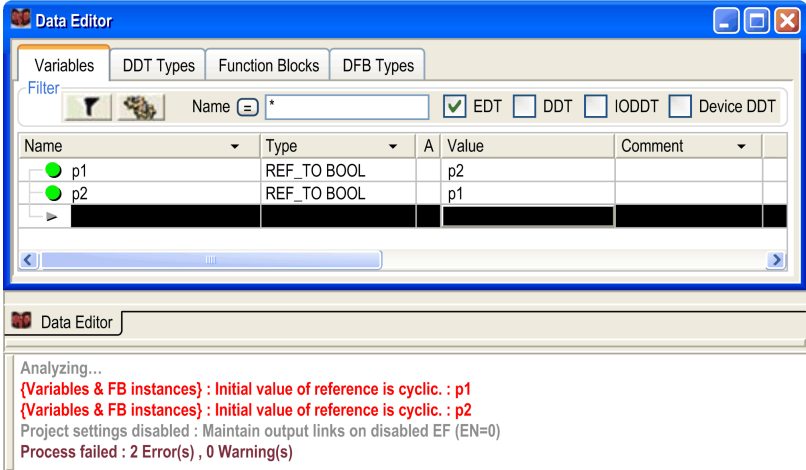
- to a reference is not supported
- cannot be explicitly assigned the NULL value
- to an IODDT is not supported because it has no memory allocation; it has no address to reference
- can only refer to variables of the given reference data type (EDT, DDT, or Device DDT) and can only be compared to a reference of the same or compatible type
- can only be used with the “:=”, “=” and “<>” operators and the EFs “EQ” and “NE”.
- cannot be a temporary variable, for example, a FBD-link or result value of a nested EF-call
- cannot be used with the SFC and LL984 programming languages
- respects the access rights of the referenced variable by variable attribute **R/W Rights of Referenced Variable**
- has to be assigned to an FFB’s reference pin (mandatory parameter)

Declaring a DFB or FFB with an input or output parameter references is allowed, but not an in/out parameter, which is already a reference.

A dereferenced reference can be used like a variable of the referenced type.

Only 1 level of dereferencing is allowed.

The initial value of a reference cannot be cyclic:



Possible usages in an application section

We can only reference an application variable to an application variable reference or to a DFB public variable reference:

- `Var_Ref := REF (Var) ;`
- `DFB_Instance.public_Var_Ref := REF (Var) ;`

We can only assign an application variable to an application variable or to a DFB public variable:

- `Var1_Ref := Var2_Ref ;`
- `DFB_Instance.public_Var_Ref := Var_Ref ;`

Possible usages in a DFB section

We can only reference an In/Out variable or a private variable, to an Out reference or Public reference for the In/Out and to a Private reference for a Private variable:

- `Out_Var_Ref := REF (In_Out_Var) ;`
- `Public_Var_Ref := REF (In_Out_Var) ;`
- `Private_Var1 := REF (Private_Var2) ;`

We can only assign an In reference, an out reference, an In/Out reference and a Public reference to an Out reference or a Public reference. And a Private references can only be assigned to a Private reference:

- `Out_Var_Ref := In_Var_Ref;`
- `Out_Var_Ref := Out_Var_Ref;`
- `Out_Var_Ref := In_Out_Var_Ref;`
- `Out_Var_Ref := Public_Var_Ref;`
- `Public_Var_Ref := In_Var_Ref;`
- `Public_Var_Ref := Out_Var_Ref;`
- `Public_Var_Ref := In_Out_Var_Ref;`
- `Public_Var_Ref := Public_Var_Ref;`
- `Private_Var_Ref := Private_Var_Ref;`

Reference access rights

The following attributes can be set to a Reference by the **Data Editor**:

- **RW program**: used to set the reference as read only.
- **R/W Rights of Referenced Variable**: used to specify if the referenced variable is a read-only variable (the referenced variable is read-only when **R/W Rights of Referenced Variable** is not selected).
- **Constant**: used to prevent modification by program.

NOTE: A reference variable has to respect the R/W attributes of the referenced variable.

This table shows the only available access rights for variables and their referenced variables:

| Reference | | | Variable | Assignment example |
|------------|----------------------------------|----------|------------|---------------------------------|
| RW Program | RW Rights of Referenced Variable | Constant | RW Program | |
| RO | RW | Yes | RO | <code>MyREF^ := Var;</code> |
| RO | RW | Yes | RW | <code>MyREF^ := Var;</code> |
| RW | RO | No | RO | <code>MyREF := REF(Var);</code> |
| RW | RO | No | RW | <code>MyREF := REF(Var);</code> |
| RW | RW | No | RO | <code>MyREF^ := Var;</code> |
| RW | RW | No | RW | <code>MyREF := REF(Var);</code> |
| RW | RW | No | RW | <code>MyREF^ := Var;</code> |

NOTE: In all other cases, Control Expert software raises a detected error, the detected error message explains how to correct the application.

Data Instances

What's in This Chapter

| | |
|--|-----|
| Data Type Instances | 226 |
| Data Instance Attributes | 230 |
| Direct Addressing Data Instances | 233 |

What's in this Chapter?

This chapter describes data instances and their characteristics.

These instances can be:

- unlocated data instances
- located data instances
- direct addressing data instances

Data Type Instances

Introduction

What is a data type instance?, page 162

A data type instance is referenced either by:

- **a name (symbol)**, in which case we say the data is **unlocated** because its memory allocation is not defined but is carried out automatically by the system,
- **a name (symbol)** and **a topological address** defined by the manufacturer, in which case we say the data is **located** since its memory allocation is known,
- **a topological address** defined by the manufacturer, in which case we say the data is **direct addressing**, and its memory allocation is known.

Unlocated Data Instances

Unlocated data instances are managed by the PLC operating system, and their physical location in the memory is unknown to the user.

Unlocated data instances are defined using data types belonging to one of the following families:

- Elementary Data Types (EDT)

- Derived Data Types (DDT)
- Device Derived Data Type (Device DDT)
- Function Block data types (EFB\DFB)
- Sequential Function Chart data types (SFC)

Examples:

```
Var_1: BOOL
;Instance of EDT family of Boolean type with 1 byte memory allocation

Var_2: UDINT
;Instance of EDT family of double unsigned integer type with 4 byte
memory allocation

Var_3: ARRAY[1..10]OF INT
;Instance of DDT family of table type with 20 byte memory allocation

COORD
  X: INT
  Y: INT
Var_4: COORD
;Instance of DDT family of COORD type structure with 4 byte memory
allocation
```

NOTE: Sequential Function Chart (SFC) data type instances are created when they are inserted in the application program, with a default name that the user can modify.

Located Data Instances

Localizing a variable (defined by a symbol) consists in creating an address in the variable editor.

Located data instances have a predefined memory location in the PLC, and this location is known by the user:

- Topological address for input/output modules
- Global address (M340, Premium) or State RAM (M580, M340, Quantum)

Located data instances are defined using data types belonging to one of the following families:

- Elementary Data Types (EDT)
- Derived Data Types (DDT)
- Input/Output Derived Data Types (IODDT)

The list below shows the data instances that should be located on a %MW, %KW addresses type:

- INT,

- UINT,
- WORD,
- BYTE,
- DATE,
- DT,
- STRING,
- TIME,
- TOD,
- DDT structure type,
- Table.

EBOOL or EBOOL tables, datas instances have to be located on a %M , %Q or %I addresses type.

IODDT datas instances type have to be located by %CH module channel type.

NOTE: Double-type instances of located data (DINT, DUNIT, DWORD) or floating (REAL) should be located by %MW, %KW addresses type. Only I/O objects instances type localization is possible with %MD<i>, %KD<i>, %QD, %ID, %MF<i>, %KF<i>, %QF, %IF type by using their topological address (for example %MD0.6.0.11, %MF0.6.0.31).

NOTE: For M580 and M340, verify that the index (i) value is even, page 199 for double-type instances of located data (%MW and %KW).

Examples:

```
Var_1 : EBOOL AT %M100
;Instance of EDT family of Boolean type (with 1 byte memory
allocation) predefined in %M100
```

```
Var_2: BOOL AT %I2.1.0.ERR
;Instance of EDT family of Boolean type (with 1 byte memory
allocation) predefined in %I2.1.0.ERR
```

```
Var_3: INT AT %MW10
;Instance of EDT family of integer type (with 2 byte memory
allocation) predefined in %MW10
```

```
Var_4: : DINT AT %MW1
;Prohibited for Modicon M340. Double type located data instances must
have a topological address even (%MW2, %MW10.....).
```

```
Var_5: WORD AT %MW10
;Instance of EDT family of WORD type (with 2 byte memory allocation)
predefined in %MW10
```

```
Var_6: ARRAY[1..10]OF INT AT %MW50
;Instance of EDT family of table type (with 20 byte memory
allocation) predefined from %MW50
```

```
COORD
  X: INT
  Y: INT
```

```
Var_7: COORD AT %MW20
;Instance of DDT family of COORD structure type (with 4 byte memory
allocation) predefined from %MW20
```

```
Var_8: DINT AT %MD0.6.0.11
;Instance of EDT family of DINT type (with 4 byte memory allocation)
predefined from the topologic address of the I/O object of the
application-specific module.
```

```
Var_9: REAL AT %MF0.6.0.31
;Instance of EDT family of REAL type (with 4 byte memory allocation)
predefined from the topologic address of the I/O object of the
application-specific module.
```

NOTE: Sequential Function Chart (SFC) data type instances are created the moment they are inserted in the application program, with a default name that the user can modify.

Direct Addressing Data Instances

Direct addressing data instances have a predefined location in the PLC memory or in an application-specific module, and this location is known to the user.

Direct addressing data instances are defined using types belonging to the Elementary Data Type (EDT) family.

Examples of direct addressing data instances:

| Internal | Constant | System | Input/Output | Network |
|------------------------------------|------------|--------|--------------|---------|
| %M<i> | | %S<i> | %Q, %I | |
| %MW<i> | %KW<i> | %SW<i> | %QW, %IW | %NW |
| %MD<i> (1) | %KD<i> (1) | | %QD, %ID | |
| %MF<i> (1) | %KF<i> (1) | | %QF, %IF | |
| Legend: | | | | |
| (1) Not available for Modicon M340 | | | | |

NOTE: Located data instances can be used by a direct addressing in the program

Example:

- Var_1: DINT AT %MW10
;%MW10 and %MW11 are both used. %MD10 direct addressing can be used or Var_1 in the program.

Data Instance Attributes

At a Glance

The attributes of a data instance are its defining information.

This information is:

- its name, page 164 (except for the direct addressing data instances, page 233)
- its topological address (except for unlocated data type instances)

- its data type, which can belong to one of the following families:
 - Elementary Data Type (EDT)
 - Derived Data Type (DDT)
 - Device derived Data type (Device DDT)
 - Function Block data type (EFB/DFB)
 - Sequential Function Chart data type (SFC)
- an optional descriptive comment (1024 characters maximum). Authorized characters correspond to the ASCII codes 32 to 255

Name of a Data Instance

This is a symbol (32 characters maximum) automatically instantiated with a default name. This name can be modified by the user.

Certain names cannot be used, for example:

- key words used in text languages
- names of program sections
- names of data types that are predefined or chosen by the user (structures, tables)
- names of DFB/EFB data types that are predefined or chosen by the user
- names of Elementary Functions (EF) that are predefined or chosen by the user

Names of Instances Belonging to the SFC Family

The names of instances are declared implicitly while the user drafts his sequential function chart. They are default names supplied by the manufacturer which the user can modify.

Manufacturer-supplied default names:

| SFC object | Name |
|---------------------------|--|
| Step | S_<section name>_<step No.> |
| Step of Macro step | S_<section name>_<macro step No.>_<step No.> |
| Macro step | MS_<section name>_<step No.> |
| Nested macro step | MS_<section name>_<macro step No.>_<step No.> |
| Input step of Macro step | S_IN<section name>_<macro step No.> |
| Output step of Macro step | S_OUT<section name>_<macro step No.> |
| Transition | T_<section name>_<transition No.> |
| Transition of Macro step | T_<section name>_<macro step No.>_<transition No.> |

Names of Instances Belonging to the Function Block Family

Instance names are implicitly declared while the user inserts the instances into the sections of the application program. They are default names supplied by the manufacturer which **the user may modify**.

Syntax of manufacturer-supplied default names:

```
FB_<name of function block type>_<instance No.>
```

NOTE: Instance names do not include the name of the section in which the instance is used, since it can be used in different sections of the application.

Access to an Element of a DDT Family Instance

The access syntax is as follows:

```
For structure type data  
<Instance name>.<Element name>
```

```
For table type data  
<Instance name>[Element index]
```

Rule:

The maximum size of the access syntax is 1024 characters, and the possible limits of a derived data type are as follows:

- 10 nesting levels (tables/structures)
- 6 dimensions per table
- 4 digits (figures) to define the index of a table element

Access to an Element of a Device DDT Family Instance

The access syntax is as follows:

```
For structure type data  
<Instance name>.<Element name>
```

```
For table type data  
<Instance name>[Element index]
```


Rule:

The maximum size of the access syntax is 1024 characters, and the possible limits of a derived data type are as follows:

- 10 nesting levels (tables/structures)
- 6 dimensions per table
- 4 digits (figures) to define the index of a table element

Direct Addressing Data Instances

At a Glance

What is a direct addressing data instance?, page 230

Access Syntax

The syntax of a direct addressing data instance is defined by the % symbol followed by a **memory location prefix** and in certain cases some additional information.

The memory location prefix can be:

- **M**, for internal variables
- **K**, for constants (Premium, M580 and M340)
- **S**, for system variables
- **N**, for network variables
- **I**, for input variables
- **Q**, for output variables

%M Internal Variables

Access syntax:

| | Syntax | Format | Example | Program access rights |
|--------------------|-----------------|----------------|---------|-----------------------|
| Bit | %M<i> or %MX<i> | 3 bits (EBOOL) | %M1 | R/W |
| Word | %MW<i> | 16 bits (INT) | %MW10 | R/W |
| Word extracted bit | %MW<i>.<j> | 1 bit (BOOL) | %MW15.5 | R/W |
| Double word | %MD<i> (1) | 32 bits (DINT) | %MD8 | R/W |

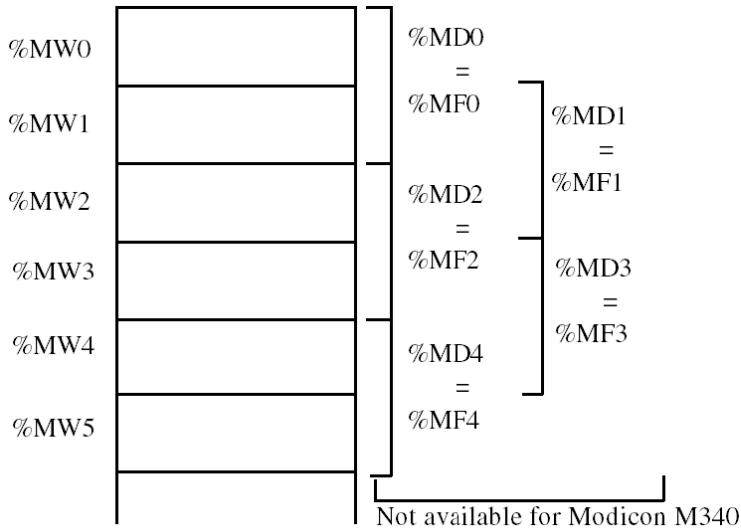
| | Syntax | Format | Example | Program access rights |
|--------------------------------------|------------|----------------|---------|-----------------------|
| Real (floating point) | %MF<i> (1) | 32 bits (REAL) | %MF15 | R/W |
| Legend | | | | |
| (1): Not available for Modicon M340. | | | | |

<i> represents the instance number (starts a 0 for Premium and 1 for Quantum).

For M580 and M340, verify that double-type instance (double word) or floating instance (real) are located in an integer type %MW and that the index <i> of the %MW is even.

NOTE: The %M<i> or %MX<i> data detect edges and manage forcing.

Memory organization:



NOTE: The modification of %MW<i> involves the corresponding modifications of %MD<i> and %MF<i>.

%K Constants

Access syntax:

| | Syntax | Format | Program access rights |
|--------------------------------------|------------|----------------|-----------------------|
| Word constant | %KW<i> | 16 bits (INT) | R |
| Double word constant | %KD<i> (1) | 32 bits (DINT) | R |
| Real (floating point) constant | %KF<i> (1) | 32 bits (REAL) | R |
| Legend | | | |
| (1): Not available for Modicon M340. | | | |

<i> represents the instance number.

NOTE: The memory organization is identical to that of internal variables, which are not available on Quantum PLCs.

%I Constants

Access syntax:

| | Syntax | Format | Program access rights |
|---------------|--------|----------------|-----------------------|
| Bit constant | %I<i> | 3 bits (EBOOL) | R |
| Word constant | %IW<i> | 16 bits (INT) | R |

<i> represents the instance number.

NOTE: These data are only available on Quantum and Momentum PLCs.

%S System Variables

Access syntax:

| | Syntax | Format | Program access rights |
|------|-----------------|---------------|-----------------------|
| Bit | %S<i> or %SX<i> | 1 bit (BOOL) | R/W or R |
| Word | %SW<i> | 32 bits (INT) | R/W or R |

<i> represents the instance number.

NOTE: The memory organization is identical to that of internal variables. The %S<i> and %SX<i> data are not used for detection of edges and do not manage forcing.

%N Network Variables

These variables contain information, which has to be exchanged between several application programs across the communication network.

Access syntax:

| | Syntax | Format | Program access rights |
|--------------------|--------------------|---------------|-----------------------|
| Common word | %NW<n>.<s>.<d> | 16 bits (INT) | R/W or R |
| Word extracted bit | %NW<n>.<s>.<d>.<j> | 1 bit (BOOL) | R/W or R |

<n> represents the network number.

<s> represents the station number.

<d> represents the data number.

<j> represents the position of the bit in the word.

Case with Input/Output Variables

These variables are contained in the application-specific modules.

Access syntax:

| | Syntax | Example | Program access rights |
|--------------------------------------|-------------------|---------------|-----------------------|
| Input/Output structure (IODDT) | %CH<@mod>.<c> | %CH4.3.2 | R |
| %I inputs | | | |
| BOOL type module detected error bit | %I<@mod>.MOD.ERR | %I4.2.MOD.ERR | R |
| BOOL type channel detected error bit | %I<@mod>.<c>.ERR | %I4.2.3.ERR | R |
| BOOL or EBOOL type bit | %I<@mod>.<c> | %I4.2.3 | R |
| | %I<@mod>.<c>.<d> | %I4.2.3.1 | R |
| INT type word | %IW<@mod>.<c> | %IW4.2.3 | R |
| | %IW<@mod>.<c>.<d> | %IW4.2.3.1 | R |
| DINT type double word | %ID<@mod>.<c> | %ID4.2.3 | R |
| | %ID<@mod>.<c>.<d> | %ID4.2.3.2 | R |
| Read type REAL (floating point) | %IF<@mod>.<c> | %IF4.2.3 | R |
| | %IF<@mod>.<c>.<d> | %IF4.2.3.2 | R |

| | Syntax | Example | Program access rights |
|--|-------------------|-------------|-----------------------|
| %Q outputs | | | |
| EBOOL type bit | %Q<@mod>.<c> | %Q4.20.3 | R/W |
| | %Q<@mod>.<c>.<d> | %Q4.20.30.1 | R/W |
| INT type word | %QW<@mod>.<c> | %QW4.2.3 | R/W |
| | %QW<@mod>.<c>.<d> | %QW4.2.3.1 | R/W |
| DINT type double word | %QD<@mod>.<c> | %QD4.2.3 | R/W |
| | %QD<@mod>.<c>.<d> | %QD4.2.3.2 | R/W |
| Read type REAL (floating point) | %QF<@mod>.<c> | %QF4.2.3 | R/W |
| | %QF<@mod>.<c>.<d> | %QF4.2.3.2 | R/W |
| %M variables (Premium) | | | |
| INT type word | %MW<@mod>.<c> | %MW4.2.3 | R/W |
| | %MW<@mod>.<c>.<d> | %MW4.2.3.1 | R/W |
| DINT type double word | %MD<@mod>.<c> | %MD4.2.3 | R/W |
| | %MD<@mod>.<c>.<d> | %MD4.2.3.2 | R/W |
| Read type REAL (floating point) | %MF<@mod>.<c> | %MF4.2.3 | R/W |
| | %MF<@mod>.<c>.<d> | %MF4.2.3.2 | R/W |
| %K Constants (Modicon M580, Modicon M340 and Premium) | | | |
| INT type word | %KW<@mod>.<c> | %KW4.2.3 | R |
| | %KW<@mod>.<c>.<d> | %KW4.2.3.1 | R |
| DINT type double word | %KD<@mod>.<c> | %KD4.2.3 | R |
| | %KD<@mod>.<c>.<d> | %KD4.2.3.12 | R |
| Read type REAL (floating point) | %KF<@mod>.<c> | %KF4.2.3 | R |
| | %KF<@mod>.<c>.<d> | %KF4.2.3.12 | R |

<@mod = \.<e>\<r>.<m>

 bus number (omitted if station is local).

<e> device connection point number (omitted if station is local, the connection point is also called Drop for Quantum users).

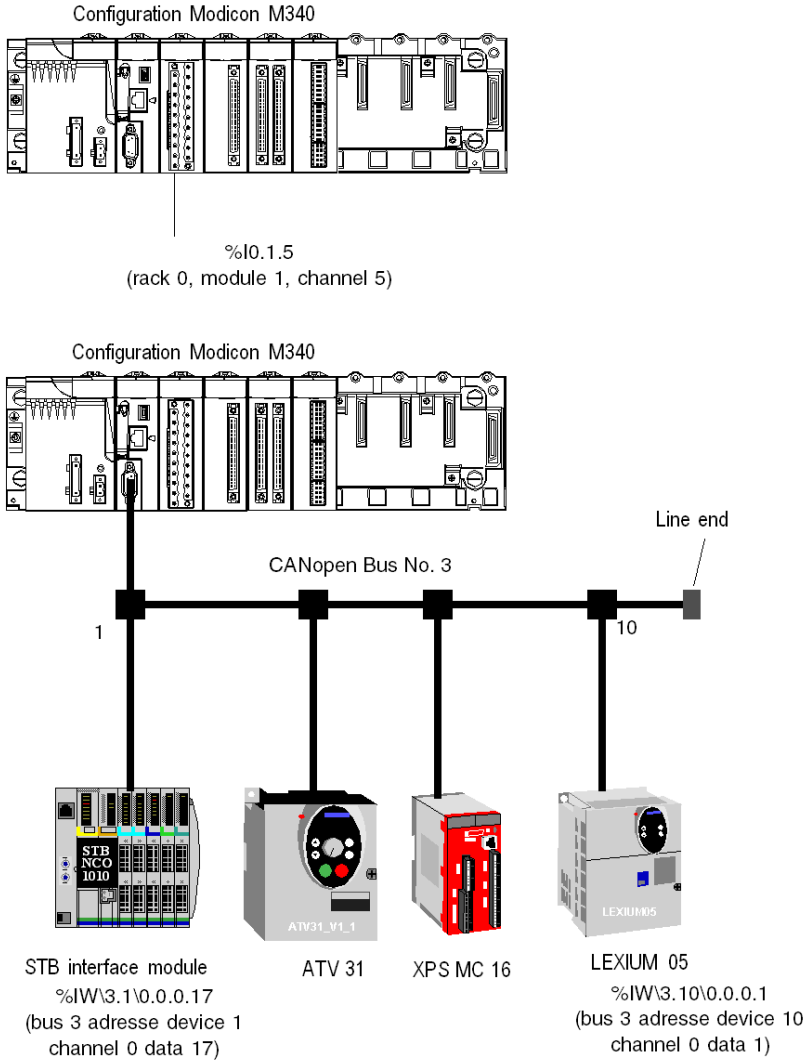
<r> rack number.

<m> module slot

<c> channel number (0 to 999) or MOD reserved word.

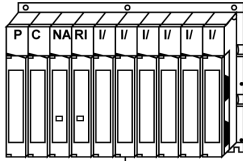
<d> data number (0 to 999) or ERR reserved word (optional if 0 value). For M580 and M340, <d> is even.

Examples: local station and station on bus for Modicon M340 PLCs.



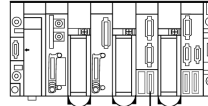
Examples: local station and station on bus for Quantum and Premium PLCs.

Quantum example

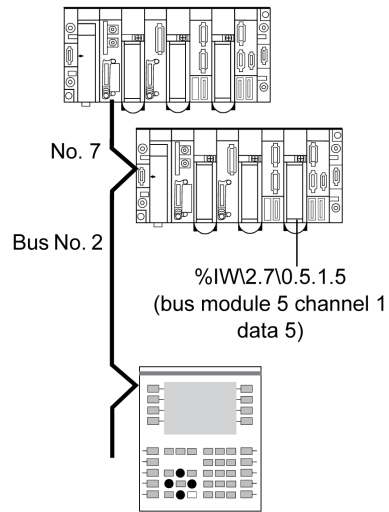
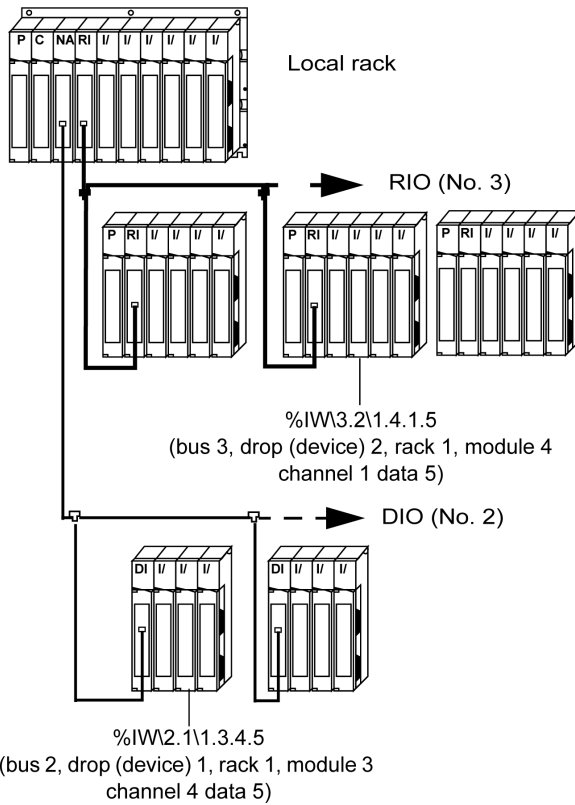


%IW1.4.1.5
(rack 1, module 4, channel 1, data 5)

Premium example



%IW0.4.1.5
(rack 0, module 4, channel 1, data 5)



Data References

What's in This Chapter

| | |
|--|-----|
| References to Data Instances by Value | 241 |
| References to Data Instances by Name | 243 |
| References to Data Instances by Address..... | 245 |
| Data Naming Rules..... | 249 |

What's in this Chapter?

This chapter provides the references of data instances.

These references can be:

- value-based references,
- name-based references,
- address-based references.

References to Data Instances by Value

Introduction

What is a data instance reference?, page 163

At a Glance

A reference to a data instance by a value is an instance which does not have a name (symbol) or topological address.

This corresponds to an **immediate value** which can be assigned to a data type instance belonging to the EDT family.

Standard IEC 1131 authorizes immediate values on instances of the following data types:

- Booleans
 - BOOL
 - EBOOL
- integers
 - INT

- UINT
- DINT
- UDINT
- TIME
- reals
 - REAL
- dates and times
 - DATE
 - DATE AND TIME
 - TIME OF DAY
- character strings
 - STRING

The programming software goes beyond the scope of the standard by adding the **bit string types**.

- BYTE
- WORD
- DWORD

Examples of Immediate Values:

This table associates immediate values with types of instance

| Immediate value | Type of instance |
|--------------------------------------|------------------|
| 'I am a character string' | STRING |
| T#1s | TIME |
| D#2000-01-01 | DATE |
| TOD#12:25:23 | TIME_OF_DAY |
| DT#2000-01-01-12:25:23 | DATE_AND_TIME |
| 16#FFF0 | WORD |
| UINT#16#9AF (typed value) | UINT |
| DWORD#16#FFFF (typed value) | DWORD |

References to Data Instances by Name

Introduction

What is a data instance reference?, page 163

References to Instances of the EDT Family

The user chooses a name (symbol) which can be used to access the data instance:

```
Valve_State: BOOL
```

```
Upper_Threshold: EBOOL AT %M10
```

```
Hopper_Content: UINT
```

```
Oven_Temperature: INT AT %MW100
```

```
Encoder_Value: WORD
```

References to Instances of the DDT Family

Tables:

The user chooses a name (symbol) which can be used to access the data instance:

Giving 2 types of table:

```
Color_Range ARRAY[1..15]OF STRING  
Vehicles ARRAY[1..100]OF Color_range  
;
```

```
Car: Vehicles  
Instance name of the Vehicles-type table
```

```
Car[11, 5]  
Access to car 11 of the color corresponding to element 5 of  
the Color_range table
```

Structures:

The user chooses a name (symbol) which can be used to access the data instance:

Giving the 2 structures:

ADDRESS

```
Street: STRING[20]
Post_Code: UDINT
Town: STRING: [20]
```

IDENT

```
Surname: STRING[15]
First name: STRING[15]
Age: UINT
Date_of_Birth: DATE
Location: ADDRESS
```

Person_1 :IDENT

;Instance name of structure of IDENT type

Person_1.Age

Access to the age of Person_1

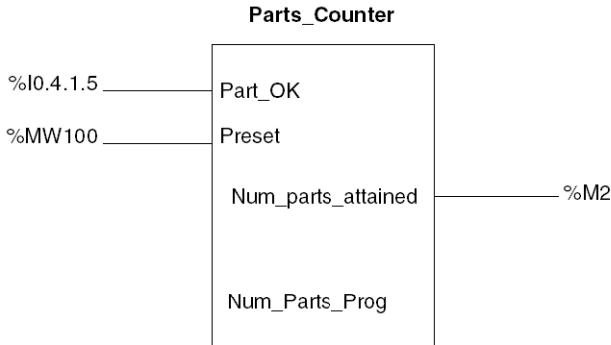
Person_1.Location.Town

;Access to the location where Person_1 resides

References to Instances of the DFB\EFB Families

The user chooses a name (symbol) which can be used to access the data instance.

Giving the DFB type:



Screw_Counter: Parts_Counter
Instance name of block of Parts_Counter type

Screw_Counter.Num_Parts_Prog
Access to public variable Num_Parts_Prog

Screw_Counter.Num_parts_attained
Access to the output interface Num_parts_attained

References to Data Instances by Address

Introduction

What is a data instance reference?, page 163

At a Glance

It is only possible to reference a data instance by address for certain data instances that belong to the **EDT family**. These instances are:

- internal variables (`%M<i>`, `%MW<i>`, `%MD<i>`, `%MF<i>`)
- constants (`%KW<i>`, `%KD<i>`, `%KF<i>`)
- inputs/outputs (`%I<address>`, `%Q<address>`)

NOTE: Instances %MD<i>, %MF<i>, %KD<i>, and %KF<i> are not available for Modicon M340 and Modicon M580.

Reference by Direct Addressing

Addressing is considered direct when the address of the instance is fixed, or, in other words, when it is written into the program.

Examples:

%M1

Access to first bit of the memory

%MW12

Access to twelfth word of the memory

%MD4

Access to fourth double word of the memory

%KF100

;Access to hundredth floating pointing word of the memory

%Q0.4.0.5

Access to fifth bit of the output module in position 4
of rack 0

References by Indexed Address

Addressing is considered indexed when the address of the instance is completed with an index.

The index is defined either by:

- a value belonging to an Integer type
- an arithmetical expression made up of Integer types

An indexed variable always has a non-indexed equivalent:

`%MW<i>[<index>] <=> %MW<j>`

The rules for calculating <j> are as follows.

| Object<i>[index] | Object<j> |
|------------------|-------------------------|
| %M<i>[index] | <j>=<i> + <index> |
| %MW<i>[index] | <j>=<i> + <index> |
| %KW<i>[index] | <j>=<i> + <index> |
| %MD<i>[index] | <j>=<i> + (<index> x 2) |
| %KD<i>[index] | <j>=<i> + (<index> x 2) |
| %MF<i>[index] | <j>=<i> + (<index> x 2) |
| %KF<i>[index] | <j>=<i> + (<index> x 2) |

Examples:

```
%MD6[10] <=> %MD26
```

```
%MW10[My_Var+8] <=> %MW20 (with My_Var=2)
```

During compilation of the program, a check verifies that:

- the index is not negative
- the index does not exceed the space in the memory allocated to each of these three data types

Word Extract Bits

It is possible to extract one of the 16 bits of single words (%MW, %SW; %KW, %IW, %QW).

The address of the instance is completed with the rank of the extracted bit (<j>).

```
WORD<i> . <j>
```

Examples:

%MW10.4

Bit No. 4 of word %MW10

%SW8.4

Bit No. 4 of system word %SW8

%KW100.14

Bit No. 14 of constant KW100

%QW0.5.1.0.10

Bit No. 10 of word 0 of channel 1 of output module 5 of rack 0

Byte Extract Bits

It is possible to extract one of the bits of a byte

The address of the extracted bit is accessible via:

- The name of the corresponding byte.
- The rank defining its position in the byte. (a number between 0 and 7)

Example:

MyByte is a variable of type BYTE. MyByte.i is a valid BOOL if $0 \leq i \leq 7$

MyByte.0, MyByte.3 and MyByte.7 are valid BOOL.

MyByte.8 is invalid.

Creating a Structure type with extracted bit

The user can create structure type using extracted bit (see EcoStruxure™ Control Expert, Operating Modes).

The **Bit Rank** dialog box is accessible by right clicking on the instance or data type which type must be:

- WORD
- UINT
- INT
- BYTE
- an extracted bit with a compatible parent

Bit and Word Tables

These are a series of adjacent objects (bits or words) of the same type and of a defined length.

OBJECT<i> :L

Presentation of bit tables:

| Type | Address | Write access |
|--------------------------|-----------|--------------|
| Discrete I/O input bits | %Ix.<i>:L | No |
| Discrete I/O output bits | %Qx.<i>:L | Yes |
| Internal bits | %M<i>:L | Yes |

Presentation of word tables:

| Type | Address | Write access |
|----------------|----------------------------------|--------------|
| Internal words | %MW<i>:L %MD<i>:L %MF<i>:L | Yes |
| Constant words | %KW<i>:L %KD<i>:L %KF<i>:L | No |
| System words | %SW50:4 | Yes |

Examples:

- %M2:65 Defines an EBOOL table from %M2 to %M66
- %M125:30 Defines an INT table from %MW125 to %MW154

Data Naming Rules

Introduction

In an application the user chooses a name to:

- define a type of data
- instantiate a data item (symbol)

- identify a Program Unit
- identify a section

Some rules have been defined in order to avoid conflicts occurring. This means that it is necessary to differentiate between the different domains of application of data

What is a Domain?

It is an area of the application from which a variable can or cannot be accessed, such as:

- the application domain which includes:
 - the various application tasks
 - the Program Units and/or sections of which it is composed
- the domains for each data type such as:
 - structures/tables for the DDT family
 - EFB/DFBs for the function block family

Rules

This table defines whether or not it is possible to use a **name** that already exists in the application for newly-created elements:

| Application Content -> New elements (below) | Program unit | Section | SR | DDT/IODDT | FB type | FB Instances | EF | Variable |
|--|--------------|-------------------|----------------|-----------|-------------------|----------------|-------------------|----------------|
| Program Unit | No | No | No | Yes | Yes | Yes | Yes | Yes |
| Section | No | No ⁽⁵⁾ | No | Yes | Yes | Yes | Yes | Yes |
| SR | No | No | No | Yes | Yes | No | ⁽¹⁾ | No |
| DDT/IODDT | No | No | No | No | No ⁽⁴⁾ | No | No ⁽⁴⁾ | No |
| FB type | Yes | Yes | Yes | No | No | ⁽³⁾ | No | ⁽³⁾ |
| FB Instances | No | No | No | No | Yes | No | Yes | No |
| EF | Yes | Yes | ⁽²⁾ | No | No | No | No | No |

| Application Content -> New elements (below) | Program unit | Section | SR | DDT/IODDT | FB type | FB Instances | EF | Variable |
|--|--------------|---------|----|-----------|---------|--------------|-----|----------|
| Variable | Yes | Yes | No | Yes | Yes | No | (1) | No |
| <p>(1) An instance belonging to the application domain cannot have the same name as an EF.</p> <p>(2) An instance belonging to the type domain (internal variable) can have the same name as an EF. The EF in question cannot be used in this type.</p> <p>(3) The creation or import of EFB/DFBs with the same name as an existing instance are prohibited.</p> <p>(4) An DDT/IODDT element might have the same name of an FB/EF, however in this case you could not use the FB/EF in the application.</p> <p>(5) Exception: there is no conflict between the name of a section belonging a program unit and the name of a section belonging to another program unit.</p> | | | | | | | | |

NOTE: A number of additional considerations to the rules given in the table are listed below, specifying that:

- Within a type, an instance (internal variable) cannot have the same name as the type name of the object to which it belongs,
- There is no conflict between the name of an instance belonging to a section of the application and the name of the instance belonging to a section of a DFB,
- There is no conflict between the name of a section belonging to a task and the name of the section belonging to a DFB.

Programming Language

What's in This Part

| | |
|----------------------------------|-----|
| Function Block Language FBD..... | 253 |
| Ladder Diagram (LD) | 278 |
| SFC Sequence Language | 319 |
| Instruction List (IL) | 374 |
| Structured Text (ST)..... | 420 |

Contents of this Part

This part describes the syntax of the programming languages that are available.

Function Block Language FBD

What's in This Chapter

| | |
|---|-----|
| General Information about the FBD Function Block Language | 253 |
| Elementary Functions, Elementary Function Blocks, Derived Function Blocks and Procedures (FFBs) | 255 |
| Subroutine Calls | 265 |
| Control Elements | 266 |
| Link | 267 |
| Text Object | 269 |
| Execution Sequence of the FFBs | 269 |
| Change Execution Sequence | 271 |
| Loop Planning | 276 |

Overview

This chapter describes the function block language FBD which conforms to IEC 61131.

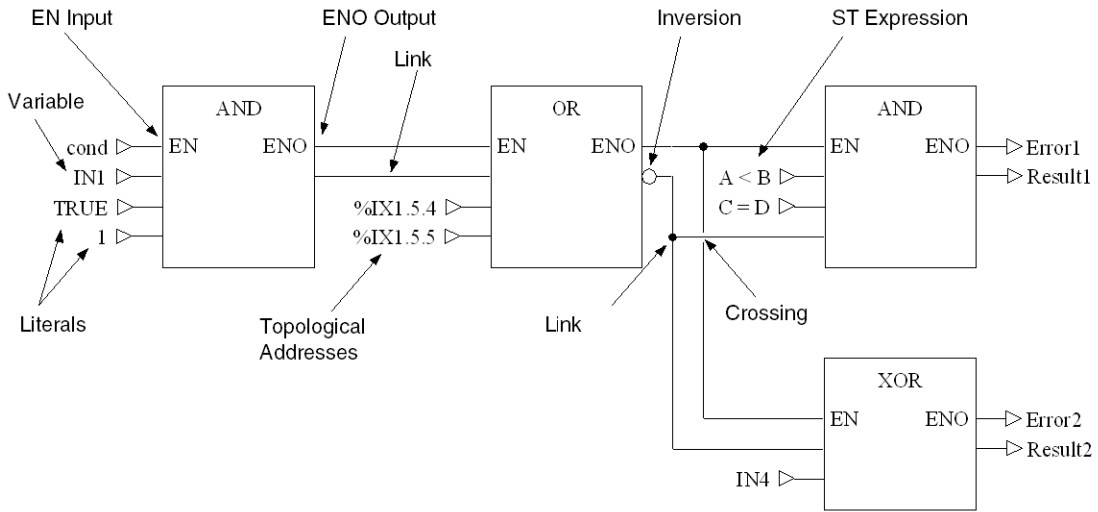
General Information about the FBD Function Block Language

Introduction

The FBD editor is used for graphical function block programming according to IEC 61131-3.

Representation of an FBD Section

Representation:



Objects

The objects of the FBD programming language (Function Block Diagram) help to divide a section into a number of:

- EFs and EFBs (Elementary Functions, page 255 and Elementary Function Blocks, page 256),
- DFBs (Derived Function Blocks), page 257,
- Procedures, page 257 and
- Control Elements, page 266.

These objects, combined under the name FFBs, can be linked with each other by:

- Links, page 267 or
- Actual Parameters, page 258.

Comments regarding the section logic can be provided using text objects (see Text Object, page 269).

Section Size

One FBD section consists of a window containing a single page.

This page has a grid background. A grid unit consists of 10 coordinates. A grid unit is the smallest possible space between 2 objects in an FBD section.

The FBD programming language is not cell oriented but the objects are still aligned with the grid coordinates.

An FBD section can be configured in number of cells (horizontal grid coordinates and vertical grid coordinates).

IEC Conformity

For a description of the extent to which the FBD programming language conforms to IEC, see IEC Conformity, page 508.

Elementary Functions, Elementary Function Blocks, Derived Function Blocks and Procedures (FFBs)

Introduction

FFB is the generic term for:

- Elementary Function (EF), page 255
- Elementary Function Block (EFB), page 256
- DFB (Derived Function Block), page 257
- Procedure, page 257

Elementary Function

Elementary functions (EF) have no internal states. If the input values are the same, the value on the output is the same every time the function is called. For example, the addition of two values always gives the same result.

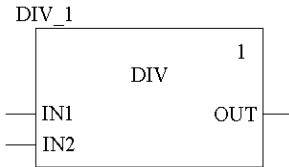
An elementary function is represented graphically as a frame with inputs and one output. The inputs are always represented on the left and the output is always on the right of the frame.

The name of the function, i.e. the function type, is displayed in the center of the frame.

The execution number, page 269 for the function is shown to the right of the function type.

The function counter is shown above the frame. The function counter is the sequential number of the function within the current section. Function counters cannot be modified.

Elementary Function



With some elementary functions, the number of inputs can be increased.

Elementary Function Block

Elementary function blocks (EFBs) have internal states. If the input values are the same, the value on the output can be different each time the function is called. e.g. for a counter the value on the output is incremented.

An elementary function block is represented graphically as a frame with inputs and outputs. The inputs are always represented on the left and the outputs always on the right of the frame.

Function blocks can have more than one output.

The name of the function block, i.e. the function block type, is displayed in the center of the frame.

The *execution number*, page 269 for the function block is shown to the right of the function block type.

The instance name is displayed above the frame.

The instance name serves as a unique identification for the function block in a project.

The EFB instance name is created automatically and has the following structure: `TYPE_n`, where:

- `TYPE` is the type of the function block.
- `n` is the sequential number of the function block type in the project.

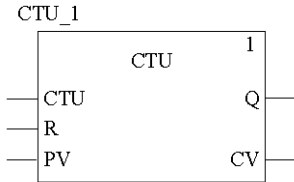
For example:

- First instance of a type EFB type TON is named `TON_0`
- First instance of a type EFB type MOTOR is named `MOTOR_0`
- Second instance of a type EFB type TON is named `TON_1`

This automatically generated name can be modified for clarification. The instance name (max. 32 characters) must be unique throughout the project and is not case-sensitive. The instance name must conform to general naming conventions.

NOTE: To conform to IEC61131-3, only letters are permitted as the first character of the name. If you want to use a numeral as your first character however, this must be enabled explicitly.

Elementary Function Block

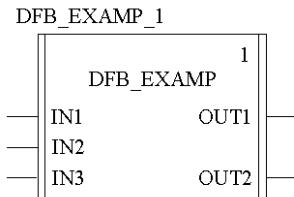


DFB

Derived function blocks (DFBs) have the same properties as elementary function blocks. The user can create them in the programming languages FBD, LD, IL, and/or ST.

The only difference to elementary function blocks is that the derived function block is represented as a frame with double vertical lines.

Derived Function Block



Procedure

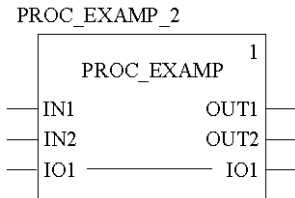
Procedures are functions viewed technically.

The only difference to elementary functions is that procedures can occupy more than one output and they support data type VAR_IN_OUT.

Procedures are a supplement to IEC 61131-3 and must be enabled explicitly.

To the eye, procedures are no different than elementary functions.

Procedure

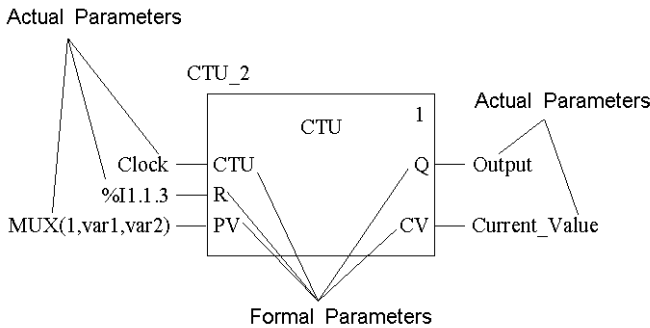


Parameters

Inputs and outputs are required to transfer values to or from an FFB. These are called formal parameters.

Objects are linked to formal parameters; these objects contain the current process states. They are called actual parameters.

Formal and actual parameters:



At program runtime, the values from the process are transferred to the FFB via the actual parameters and then output again after processing.

Only one object (actual parameter) of the following types may be linked to FFB inputs:

- Variable
- Address
- Literal
- ST Expression, page 420

ST expressions on FFB inputs are a supplement to IEC 61131-3 and must be enabled explicitly.

- Link

The following combinations of objects (actual parameters) can be linked to FFB outputs:

- one variable
- a variable and one or more connections (but not for VAR_IN_OUT, page 264 outputs)
- an address
- an address and one or more connections (but not for VAR_IN_OUT, page 264 outputs)
- one or more connections (but not for VAR_IN_OUT, page 264 outputs)

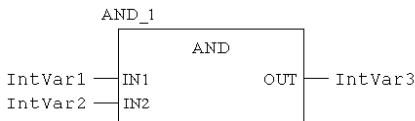
The data type of the object to be linked must be the same as that of the FFB input/output. If all actual parameters consist of literals, a suitable data type is selected for the function block.

Exception: For generic FFB inputs/outputs with data type ANY_BIT, it is possible to link objects of data type INT or DINT (not UINT and UDINT).

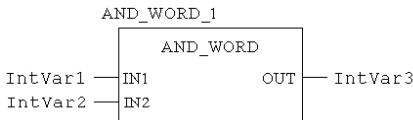
This is a supplement to IEC 61131-3 and must be enabled explicitly.

Example:

Allowed:



Not allowed:



(In this case, AND_INT must be used.)

Not all formal parameters have to be assigned an actual parameter. However, this does not apply in the case of negated pins. These must always be assigned an actual parameter. This is also the case with some formal parameter types. These types are shown in the following table.

Table of formal parameter types:

| Parameter type | EDT | STRING | ARRAY | ANY_ARRAY | IODDT | DEVICE DDT | STRUC-T | FB | ANY |
|-----------------|-----|--------|-------|-----------|-------|------------|---------|----|-----|
| EFB: Input | - | - | - | - | / | / | - | / | - |
| EFB: VAR_IN_OUT | + | - | - | - | + | / | - | / | - |
| EFB: Output | - | - | + | + | + | / | - | / | + |
| DFB: Input | - | - | - | - | / | + | - | / | - |

| Parameter type | EDT | STRING | ARRAY | ANY_ARRAY | IODDT | DEVICE DDT | STRUC-T | FB | ANY |
|--|-----|--------|-------|-----------|-------|------------|---------|----|-----|
| DFB: VAR_IN_OUT | + | - | - | - | + | + | - | / | - |
| DFB: Output | - | - | + | / | / | / | - | / | + |
| EF: Input | - | - | - | - | + | / | - | + | - |
| EF: VAR_IN_OUT | + | - | - | - | + | / | - | / | - |
| EF: Output | - | - | - | - | - | / | - | / | - |
| Procedure: Input | - | - | - | - | + | / | - | + | - |
| Procedure: VAR_IN_OUT | + | + | + | + | + | / | + | / | + |
| Procedure: Output | - | - | - | - | - | / | - | / | + |
| + Actual parameter required | | | | | | | | | |
| - Actual parameter not required, it's the general rule, but there are exceptions for some FFBs, for instance when some parameters are used to characterize the information we want to be given by the FFB. | | | | | | | | | |
| / not applicable | | | | | | | | | |

FFBs that use actual parameters on the inputs that have not yet received any value assignment, work with the initial values of these actual parameters.

If no value is allocated to a formal parameter, then the initial value is used for executing the function block. If no initial value has been defined then the default value ("0") is used.

If a formal parameter is not assigned a value and the function block/DFB instance is invoked more than once, then the subsequently executed invocations are run with the last effective actual value.

NOTE: Unassigned data structures are always initialized with value "0", initial values can not be defined.

NOTE: An ANY_ARRAY_xxx input pin not connected will create automatically an hidden array of 1 element.

Public Variables

In addition to inputs and outputs, some function blocks also provide public variables.

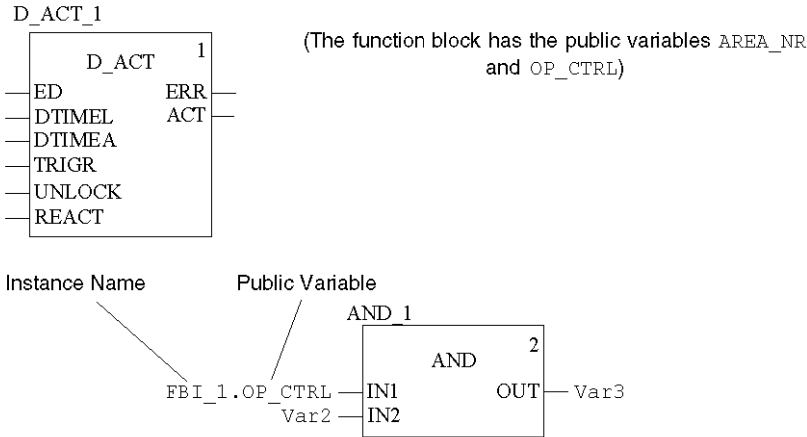
These variables transfer static values (values that are not influenced by the process) to the function block. They are used for setting parameters for the function block.

Public variables are a supplement to IEC 61131-3.

The assignment of values to public variables is made using their initial values.

Public variables are read via the instance name of the function block and the names of the public variables.

Example:



Private Variables

In addition to inputs, outputs and public variables, some function blocks also provide private variables.

Like public variables, private variables are used to transfer statistical values (values that are not influenced by the process) to the function block.

Private variables can not be accessed by user program. These type of variables can only be accessed by the animation table.

NOTE: Nested DFBs are declared as private variables of the parent DFB. So their variables are also not accessible through programming, but trough the animation table.

Private variables are a supplement to IEC 61131-3.

Programming Notes

Attention should be paid to the following programming notes:

- FFBs are only executed if the input `EN=1` or if the input `EN`, page 262 is grayed out.
- Boolean inputs and outputs can be inverted.
- Special conditions apply when using `VAR_IN_OUT` variables, page 264.
- Function block/DFB instances can be called multiple times, page 262.

Multiple Function Block Instance Call

Function block/DFB instances can be called more than once; other than instances from communication EFBs and function blocks/DFBs with an `ANY` output but no `ANY` input: these can only be called once.

Calling the same function block/DFB instance more than once makes sense, for example, in the following cases:

- If the function block/DFB has no internal value or it is not required for further processing.
 In this case, memory is saved by calling the same function block/DFB instance more than once since the code for the function block/DFB is only loaded once.
 The function block/DFB is then handled like a "Function".
- If the function block/DFB has an internal value and this is supposed to influence various program segments, for example, the value of a counter should be increased in different parts of the program.
 In this case, calling the same function block/DFB means that temporary results do not have to be saved for further processing in another part of the program.

EN and ENO

One `EN` input and one `ENO` output can be used in all FFBs.

If the value of `EN` is equal to "0" when the FFB is invoked, the algorithms defined by the FFB are not executed and `ENO` is set to "0".

If the value of `EN` is equal to "1" when the FFB is invoked, the algorithms defined by the FFB is executed. After the algorithms have been executed successfully, the value of `ENO` is set to "1". If an error occurs when executing these algorithms, `ENO` is set to "0".

If the `EN` pin is not assigned a value, when the FFB is invoked, the algorithm defined by the FFB is executed (same as if `EN` equals to "1"), Please refer to *Maintain output links on disabled EF* (see EcoStruxure™ Control Expert, Operating Modes).

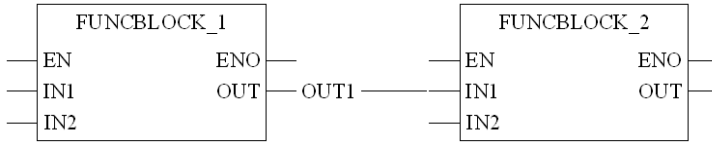
If `ENO` is set to "0" (caused by `EN=0` or an error during execution):

- Function blocks
 - `EN/ENO` handling with function blocks that (only) have one link as an output parameter:



If EN of FUNCBLOCK_1 is set to "0", the link on output OUT of FUNCBLOCK_1 maintains the old status it had during the last correctly executed cycle.

- EN/ENO handling with function blocks that have one variable and one link as output parameters:



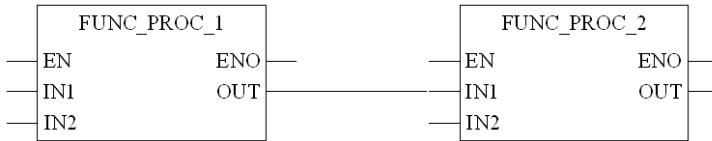
If EN of FUNCBLOCK_1 is set to "0", the link on output OUT of FUNCBLOCK_1 maintains the old status it had during the last correctly executed cycle. The OUT1 variable on the same pin either retains its previous status or can be changed externally without influencing the link. The variable and the link are saved independently of each other.

- Functions/Procedures

As defined in IEC61131-3, the outputs from deactivated functions (EN input set to "0") are undefined. (The same applies to procedures.)

Here nevertheless an explanation of the output statuses in this case:

- EN/ENO handling with function/procedure blocks that (only) have one link as an output parameter:



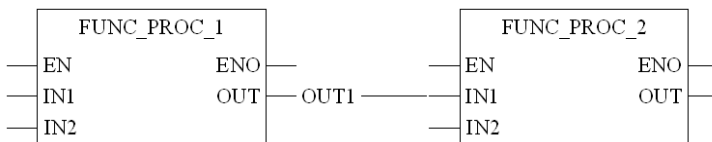
If EN of FUNC_PROC_1 is set to "0", the value of the link on output OUT of FUNC_PROC_1 depends on the project setting **Maintain output links on disabled EF**.

If this project setting is set to "0", the value of the link is set to "0".

If this project setting is set to "1", the link maintains the old value it had during the last correctly executed cycle.

Please refer to *Maintain output links on disabled EF* (see EcoStruxure™ Control Expert, Operating Modes).

- EN/ENO handling with function/procedure blocks that have one variable and one link as output parameters:



If EN of FUNC_PROC_1 is set to "0", the value of the link on output OUT of FUNC_PROC_1 depends on the project setting **Maintain output links on disabled EF**.

If this project setting is set to "0", the value of the link is set to "0".

If this project setting is set to "1", the link maintains the old value it had during the last correctly executed cycle.

Please refer to *Maintain output links on disabled EF* (see EcoStruxure™ Control Expert, Operating Modes).

The OUT1 variable on the same pin either retains its previous status or can be changed externally without influencing the link. The variable and the link are saved independently of each other.

The output behavior of the FFBs does not depend on whether the FFBs are invoked without EN/ENO or with EN=1.

NOTE: For disabled function blocks (EN = 0) with an internal time function (e.g. function block DELAY), time seems to keep running, since it is calculated with the help of a system clock and is therefore independent of the program cycle and the release of the block.

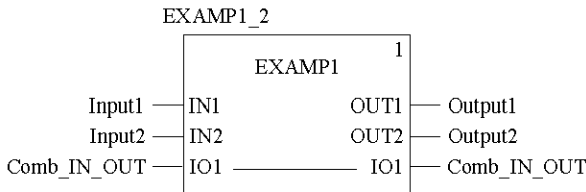
VAR_IN_OUT Variable

FFBs are often used to read a variable at an input (input variables), to process it and to output the altered values of the **same** variable (output variables).

This special type of input/output variable is also called a VAR_IN_OUT variable.

The link between input and output variables is represented by a line in the FFB.

VAR_IN_OUT variable



The following special features are to be noted when using FFBs with VAR_IN_OUT variables.

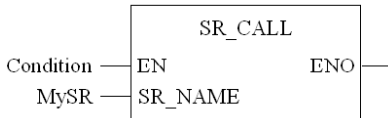
- All VAR_IN_OUT inputs must be assigned a variable.
- Via graphical links only VAR_IN_OUT outputs with VAR_IN_OUT inputs can be connected.
- Only one graphical link can be connected to a VAR_IN_OUT input/output.

- A combination of variable/address and graphical connections is not possible for VAR_IN_OUT outputs).
- No literals or constants can be connected to VAR_IN_OUT inputs/outputs.
- No negations can be used on VAR_IN_OUT inputs/outputs.
- Different variables/variable components can be connected to the VAR_IN_OUT input and the VAR_IN_OUT output. In this case the value of the variables/variable component on the input is copied to the at the output variables/variable component.

Subroutine Calls

Calling a Subroutine

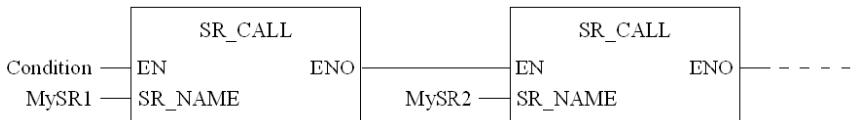
In FBD, subroutines are called using the following blocks.



If the status of EN is 1, the respective subroutine (variable name inSR_Name) is called.

The output ENO is not used to display the error status for this type of block. The output ENO is always 1 for this type of block and is used to call multiple subroutines simultaneously.

The following construction makes it possible to call multiple subroutines simultaneously.



The subroutine to be called must be located in the same task as the FBD section called.

Subroutines can also be called from within subroutines.

Subroutine calls are a supplement to IEC 61131-3 and must be enabled explicitly.

In SFC action sections, subroutine calls are only allowed when Multitoken Operation is enabled.




Control Elements

Introduction

Control elements are used for executing jumps within an FBD section and for returning from a subroutine (SRx) or derived function block (DFB) to the main program.

Control Elements

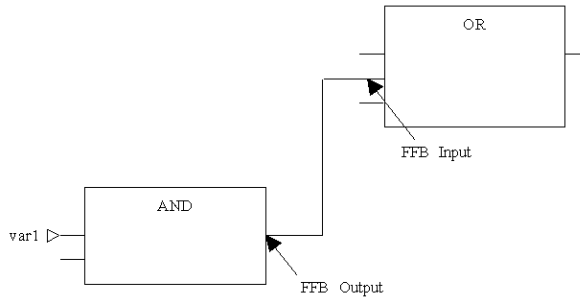
The following control elements are available.

| Designation | Representation | Description |
|-------------|---|--|
| Jump |  | <p>When the status of the left link is 1, a jump is made to a label (in the current section).</p> <p>To generate a conditional jump, a jump object is linked to a Boolean FFB output.</p> <p>To generate an unconditional jump, the jump object is assigned the value 1 for example, using the <code>AND</code> function.</p> |
| Label |  | <p>Labels (jump targets) are indicated as text with a colon at the end.</p> <p>This text is limited to 32 characters and must be unique within the entire section. The text must conform to general naming conventions.</p> <p>Jump labels can only be placed between the first two grid points on the left edge of the section.</p> <p>Note: Jump labels may not "cut through" networks, i.e. an assumed line from the jump label to the right edge of the section may not be crossed by any object. This is also valid for links.</p> |
| Return |  | <p><code>RETURN</code> objects can not be used in the main program.</p> <ul style="list-style-type: none"> In a DFB, a <code>RETURN</code> object forces the return to the program which called the DFB. <ul style="list-style-type: none"> The rest of the DFB section containing the <code>RETURN</code> object is not executed. The next sections of the DFB are not executed. <p>The program which called the DFB will be executed after return from the DFB.</p> <p>If the DFB is called by another DFB, the calling DFB will be executed after return.</p> In a SR, a <code>RETURN</code> object forces the return to the program which called the SR. <ul style="list-style-type: none"> The rest of the SR containing the <code>RETURN</code> object is not executed. <p>The program which called the SR will be executed after return from the SR.</p> |

Link

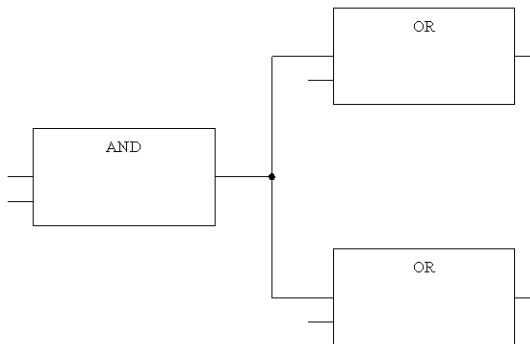
Description

Links are vertical and horizontal connections between FFBs.

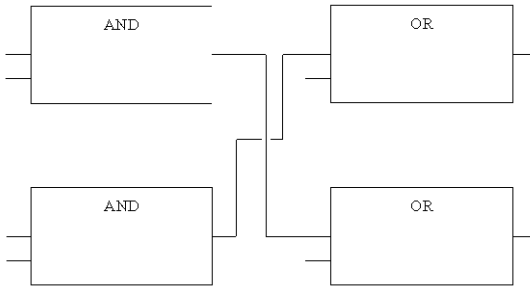


Representation

The link coordinates are identified by a filled circle.



Crossed links are indicated by a "broken" link.



Programming Notes

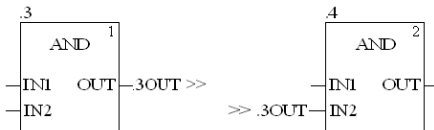
Attention should be paid to the following programming notes:

- Links can be used for any data type.
- The data types of the inputs/outputs to be linked must be the same.
- Several links can be connected with one FFB output. Only one may be linked with an FFB input however.
- Inputs and outputs may be linked to one-another. Linking more than one output together is not possible. That means that no OR connection is possible using links in FBD. An OR function is to be used in this case.
- Overlapping links with other objects is permitted.
- Links may not be used to create loops since the sequence of execution in this case cannot be clearly determined in the section. Loops must be created using actual parameters (see [Loop Planning](#), page 276).
- To avoid links crossing each other, links can also be represented in the form of connectors.

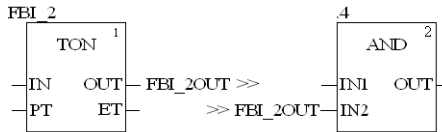
The source and target for the connection are labeled with a name that is unique within the section.

The connector name has the following structure depending on the type of source object for the connection:

- For functions: "Function counter/formal parameter" for the source of the connection



- For function blocks: "Instance name/formal parameter" for the source of the connection



Text Object

Description

Text can be positioned as text objects using FBD Function Block language. The size of these text objects depends on the length of the text. The size of the object, depending on the size of the text, can be extended vertically and horizontally to fill further grid units. Text objects may not overlap with FFBs; however they may overlap with links.

Execution Sequence of the FFBs

Introduction

The execution sequence is determined by the position of the FFBs within the section (executed from left to right and from top to bottom). If the FFBs are then linked graphically, the execution sequence is determined by the signal flow.

The execution sequence is indicated by the execution number (number in the top right corner of the FFB frame).

Execution Sequence on Networks

For network execution sequences, the following rules apply:

- Executing a section is completed network by network based on the FFB links from above and below.
- Links may not be used to create loops since the sequence of execution in this case cannot be clearly determined. Loops must be created using actual parameters (see Loop Planning, page 276).
- The execution sequence for networks that are not linked is determined by the graphic sequence (from top-right to bottom-left). This execution sequence can be influenced (see Change Execution Sequence, page 271).
- Processing on a network is ended completely before the processing begins on another network for which outputs are used on the previous network.

- No element of a network is deemed to be processed as long as the status of all inputs of this element are not calculated.
- Processing on a network is only ended if all outputs on this network have been processed.

Signal Flow within a Network

For execution sequences within a network, the following rules apply:

- An FFB is only processed if all elements (FFB outputs etc.) with which its inputs are linked are processed.
- The execution sequence of FFBs that are linked with various outputs of the same FFB runs from top to bottom.
- The execution sequence of FFBs is not influenced by the location within the network.

This does not apply if more than one FFB is linked to the same output of the "calling" FFB. In this case, the execution sequence is determined by the graphic sequence (from top to bottom).

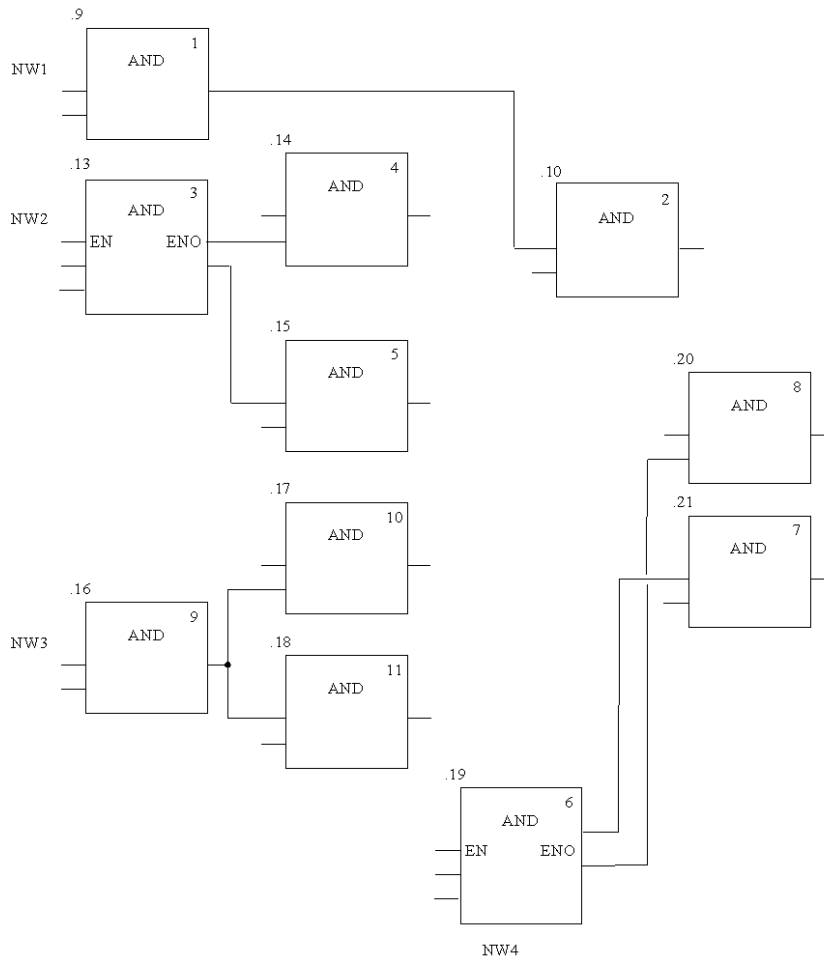
Priorities

Priorities in Defining the Signal Flow Within a Section.

| Priority | Rule | Description |
|----------|--------------------|---|
| 1 | Link | Links have the highest priorities in defining the signal flow within a FBD section. |
| 2 | User Definition | User Access to Execution Sequence. |
| 3 | Network by Network | Processing on a network is ended completely before the processing begins on another network. |
| 4 | Output Sequence | FFBs that are linked to the outputs of the same "calling" FFB are processed from top to bottom. |
| 5 | Rung by Rung | Lowest priority. (Only applies if none of the other rules apply). |

Example

Example of the Execution Sequence of Objects in an FBD Section:



Change Execution Sequence

Introduction

The execution order of networks and the execution order of objects within a network are defined by a number of rules, page 270.

In some cases the execution order suggested by the system should be changed.

The procedure for defining/changing the execution sequence of networks is as follows:

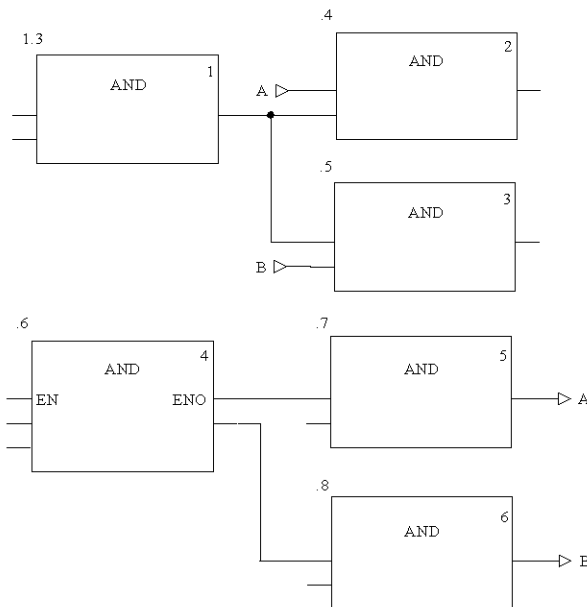
- Using links instead of actual parameters
- Network positions
- Explicit execution sequence definition

The procedure for defining/changing the execution sequence of networks is as follows:

- FFB positions

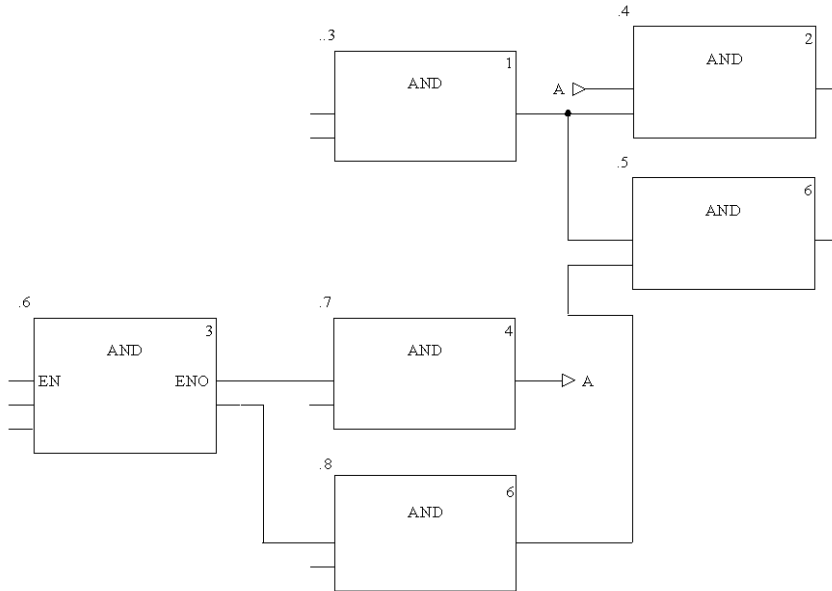
Original Situation

The following diagram shows two networks for which the execution sequences are simply defined by their positions within the section, without taking into account the fact that blocks .4/.5 and .7/.8 require a different execution sequence.



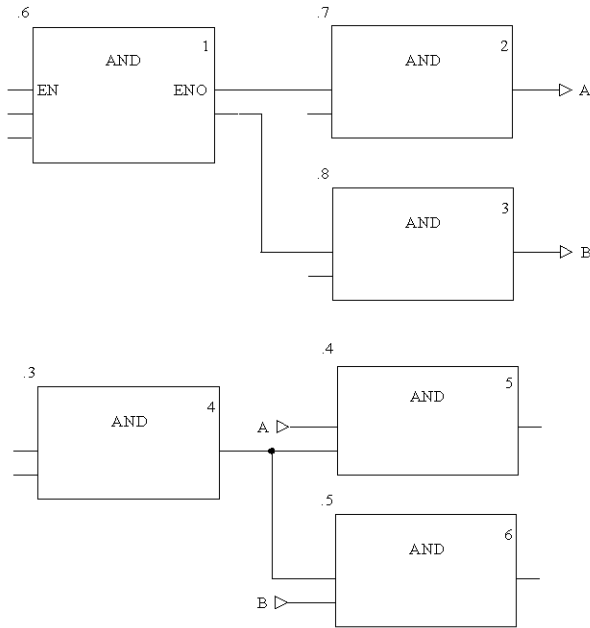
Link Instead of Actual Parameters

By using a link instead of a variable the two networks are executed in the proper sequence (see also Original Situation, page 272).



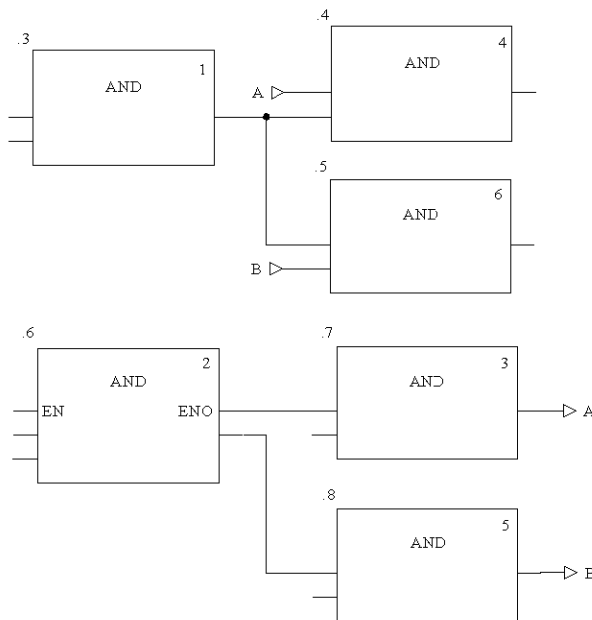
Network Positions

The correct execution sequence can be achieved by changing the position of the networks in the section (see also Original Situation, page 272).



Explicit Definition

The correct execution sequence can be achieved by explicitly changing the execution sequence of an FFB. To indicate that which FFB's had their execution order changed, the execution number is shown in a black field (see also *Original Situation*, page 272).



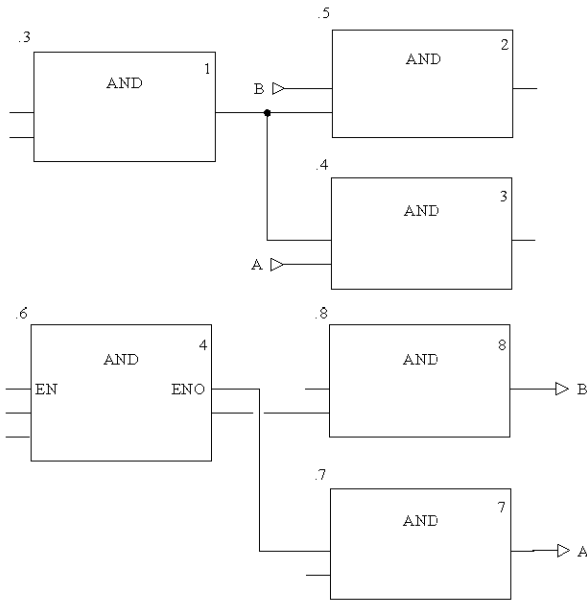
NOTE: Only one reference of an instance is allowed, e.g. the instance ".7" may only be referenced once.

FFB Positions

The position of FFBs only influences the execution sequence if more than one FFB is linked to the same output of the "calling" FFB (see also *Original Situation*, page 272).

In the first network, block positions .4 and .5 are switched. In this case (common origins for both block inputs) the execution sequence of both blocks is switched as well (processed from top to bottom).

In the second network, block positions .7 and .8 are switched. In this case (different origins for the block inputs) the execution sequence of the blocks is not switched (processed in the order the block outputs are called).

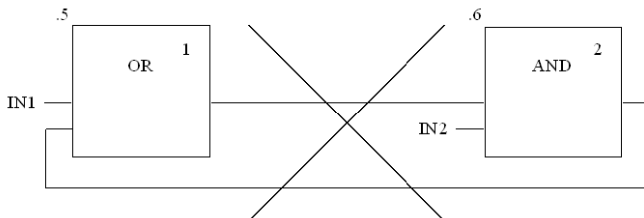


Loop Planning

Non-Permitted Loops

Configuring loops exclusively via links is not permitted since it is not possible to clearly specify the signal flow (the output of one FFB is the input of the next FFB, and the output of this one is the input of the first).

Non-permitted Loops via Links



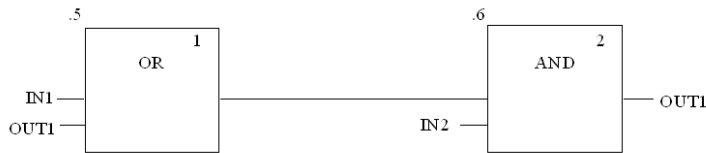
Generating Via an Actual Parameter

This type of logic must be resolved using feedback variables so that the signal flow can be determined.

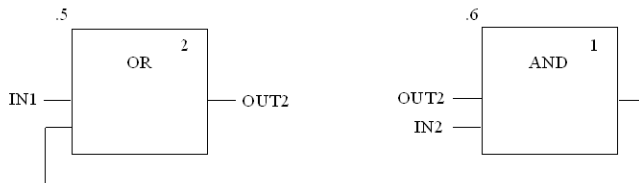
Feedback variables must be initialized. The initial value is used during the first execution of the logic. Once they have been executed the initial value is replaced by the actual value.

Pay attention to the two different types of execution sequences (number in brackets after the instance name) for the two blocks.

Loop generated with an actual parameter: Type 1



Loop generated with an actual parameter: Type 2



Ladder Diagram (LD)

What's in This Chapter

| | |
|---|-----|
| General Information about the LD Ladder Diagram Language | 278 |
| Contacts | 281 |
| Coils | 282 |
| Elementary Functions, Elementary Function Blocks, Derived Function Blocks and Procedures (FFBs) | 283 |
| Control Elements | 294 |
| Operate Blocks and Compare Blocks | 295 |
| Links | 297 |
| Text Object | 300 |
| Edge Recognition | 301 |
| Execution Sequence and Signal Flow | 309 |
| Loop Planning | 311 |
| Change Execution Sequence | 313 |

Overview

This chapter describes the ladder diagram language LD which conforms to IEC 61131.1.

General Information about the LD Ladder Diagram Language

Introduction

This section describes the Ladder Diagram (LD) according to IEC 61131-3.

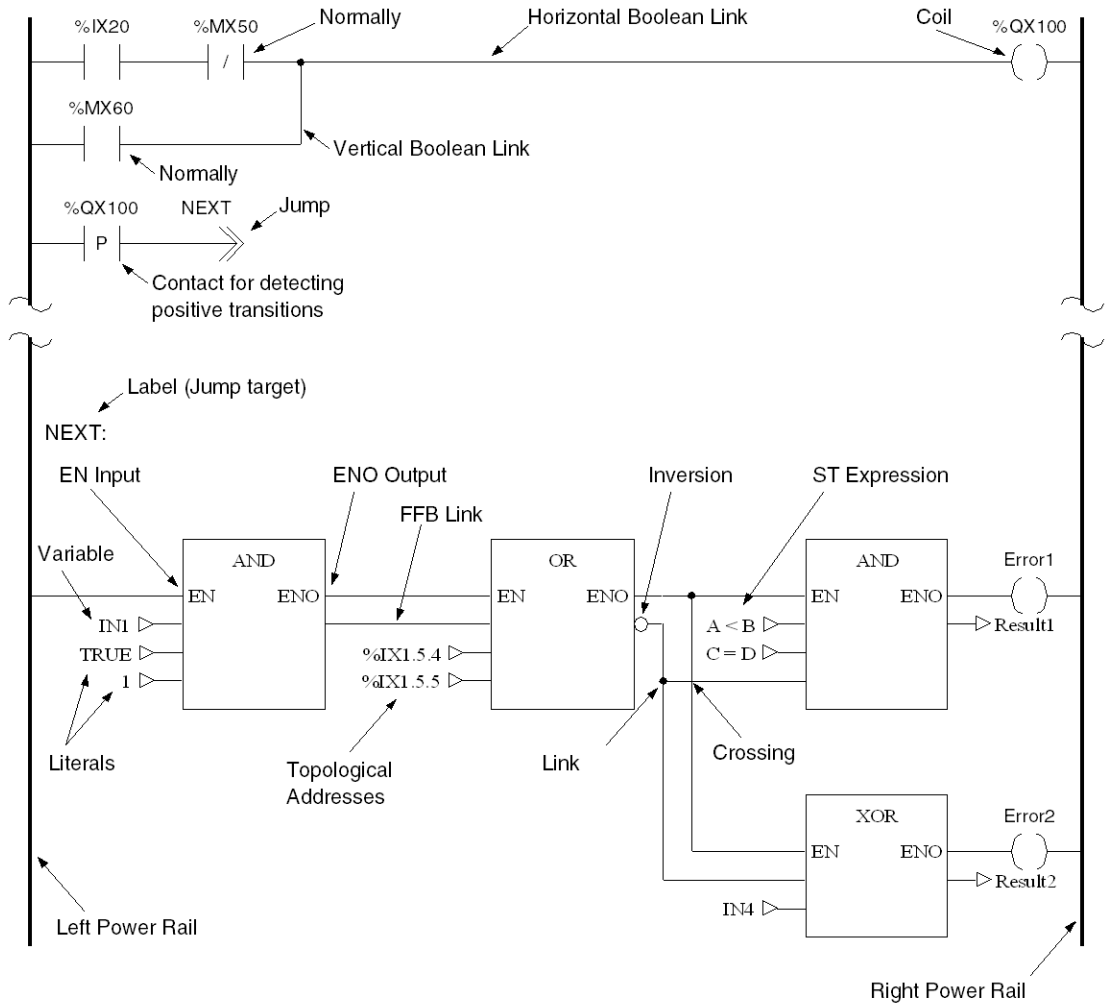
The structure of an LD section corresponds to a rung for relay switching.

The left power rail is located on the left-hand side of the LD editor. This left power rail corresponds to the phase (L ladder) of a rung. With LD programming, in the same way as in a rung, only the LD objects which are linked to a power supply, that is to say connected to the left power rail, are "processed". The right power rail corresponds to the neutral wire. However, all coils and FFB outputs are linked with it directly or indirectly, and this creates a power flow.

A group of objects which are linked together one below the other, and have no links to other objects (excluding the power rail), is called a network or a rung.

Representation of an LD Section

Representation:



Objects

The objects of the LD programming language help to divide a section into a number of:

- Contacts, page 281
- Coils, page 282

- EFs and EFBs (Elementary Functions, page 283 and Elementary Function Blocks, page 284)
- DFBs (Derived Function Blocks, page 285)
- Procedures, page 286
- Control Elements, page 294 and
- Operation and Comparison blocks, page 295 that represent an extension to IEC 61131-3

These objects can be connected with each other by means of:

- Links, page 297 or
- Actual Parameters, page 286 (FFBs only).

Comments regarding the section logic can be provided using text objects (see Text Object, page 300).

Section Size

One LD section consists of a window containing a single page.

This page has a grid that divides the section into rows and columns.

A width of 11-63 columns and 17-3998 lines can be defined for LD sections.

The LD programming language is cell oriented, i.e. only one object can be placed in each cell.

Processing Sequence

The processing sequence of the individual objects in an LD section is determined by the data flow within the section. Networks connected to the left power rail are processed from top to bottom (link to the left power rail). Networks that are independent of each other within the section are processed according to their position (from top to bottom) (see also Execution Sequence and Signal Flow, page 309).

IEC Conformity

For a description of IEC conformity for the LD programming language, see IEC Conformity, page 508.

Contacts

Introduction

A contact is an LD element that transfers a status on the horizontal link to its right side. This status is the result of a Boolean AND operation on the status of the horizontal link on the left side with the status of the relevant Boolean actual parameter.

A contact does not change the value of the relevant actual parameter.

Contacts take up one cell.


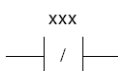
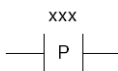
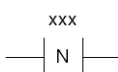
The following are permitted as actual parameters:

- Boolean variables
- Boolean constants
- Boolean addresses (topological addresses or symbolic addresses)
- ST expression, page 420 delivering a Boolean result (e.g. `VarA OR VarB`)

ST expressions as actual parameters for contacts are a supplement to IEC 61131-3 and must be enabled explicitly

Contact Types

The following contacts are available:

| Designation | Representation | Description |
|--|---|---|
| Normally open |  | In the case of normally open contacts, the status of the left link is transferred to the right link if the status of the relevant Boolean actual parameter (indicated with xxx) is ON. Otherwise, the status of the right link is OFF. |
| Normally closed |  | In the case of normally closed contacts, the status of the left link is transferred to the right link if the status of the relevant Boolean actual parameter (indicated with xxx) is OFF. Otherwise, the status of the right link is OFF. |
| Contact for detecting positive transitions |  | With contacts for detection of positive transitions, the right link for a program cycle is ON if a transfer of the relevant actual parameter (labeled by xxx) goes from OFF to ON and the status of the left link is ON at the same time. Otherwise, the status of the right link is 0. Also see Edge Recognition, page 301. |
| Contact for detecting negative transitions |  | With contacts for detection of negative transitions, the right link for a program cycle is ON if a transfer of the relevant actual parameter (labeled by xxx) goes from ON to OFF and the status of the left link is ON at the same time. Otherwise, the status of the right link is 0. Also see Edge Recognition, page 301. |

Coils

Introduction

A coil is an LD element which transfers the status of the horizontal link on the left side, unchanged, to the horizontal link on the right side. The status is stored in the respective Boolean actual parameter.

Normally, coils follow contacts or FFBs, but they can also be followed by contacts.

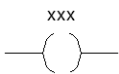
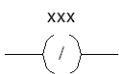
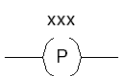
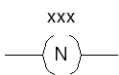
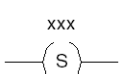
Coils take up one cell.

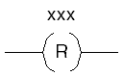
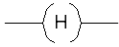
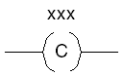
The following are permitted as actual parameters:

- Boolean variables
- Boolean addresses (topological addresses or symbolic addresses)

Coil Types

The following coils are available:

| Designation | Representation | Description |
|---|---|---|
| Coil |  | With coils, the status of the left link is transferred to the relevant Boolean actual parameter (indicated by xxx) and the right link. |
| Negated coil |  | With negated coils, the status of the left link is copied onto the right link. The inverted status of the left link is copied to the relevant Boolean actual parameter (indicated by xxx). If the left link is OFF, then the right link will also be OFF and the relevant Boolean actual parameter will be ON. |
| Coil for detecting positive transitions |  | With coils that detect positive transitions, the status of the left link is copied onto the right link. The relevant actual parameter of data type EBOOL (indicated by xxx) is 1 for a program cycle, if a transition of the left link from 0 to 1 is made. Also see Edge Recognition , page 301. |
| Coil for detecting negative transitions |  | With coils that detect negative transitions, the status of the left link is copied onto the right link. The relevant actual Boolean parameter (indicated by xxx) is 1 for a program cycle, if a transition of the left link from 1 to 0 is made. Also see Edge Recognition , page 301. |
| Set coil |  | With set coils, the status of the left link is copied onto the right link. The relevant Boolean actual parameter (indicated by xxx) is set to ON if the left link has a status of ON, otherwise it remains unchanged. The relevant Boolean actual parameter can be reset through the reset coil. Also see Edge Recognition , page 301. |

| Designation | Representation | Description |
|-------------|---|---|
| Reset coil |  | <p>With reset coils, the status of the left link is copied onto the right link. The relevant Boolean actual parameter (indicated by xxx) is set to OFF if the left link has a status of ON, otherwise it remains unchanged. The relevant Boolean actual parameter can be set through the set coil.</p> <p>Also see Edge Recognition, page 301.</p> |
| Halt coil |  | <p>With halt coils, if the status of the left link is 1, the program execution is stopped immediately (With halt coils the status of the left link is not copied to the right link.). Sets the CPU to HALT mode, page 156.</p> |
| Call coil |  | <p>With call coils, the status of the left link is copied to the right link. If the status of the left link is ON then the respective sub-program (indicated by xxx) is called.</p> <p>The subroutine to be called must be located in the same task as the calling LD section. Subroutines can also be called from within subroutines.</p> <p>Subroutines are a supplement to IEC 61131-3 and must be enabled explicitly.</p> <p>In SFC action sections, call coils (subroutine calls) are only allowed when Multitoken Operation is enabled.</p> |

Elementary Functions, Elementary Function Blocks, Derived Function Blocks and Procedures (FFBs)

Introduction

FFB is the generic term for:

- Elementary Function (EF), page 283
- Elementary Function Block (EFB), page 284
- Derived Function Block (DFB), page 285
- Procedure, page 286

FFBs occupy 1 to 3 columns (depending on the length of the formal parameter names) and 2 to 33 lines (depending on the number of formal parameter rows).

Elementary Function

Functions have no internal states. If the input values are the same, the value on the output is the same every time the function is called. For example, the addition of two values always gives the same result.

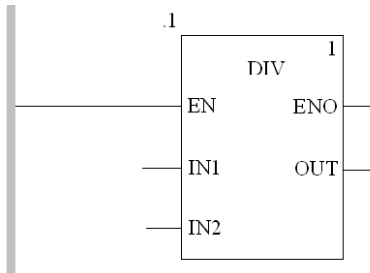
An elementary function is represented graphically as a frame with inputs and one output. The inputs are always represented on the left and the output is always on the right of the frame.

The name of the function, i.e. the function type, is displayed in the center of the frame.

The execution number, page 309 for the function is shown to the right of the function type.

The function counter is shown above the frame. The function counter is the sequential number of the function within the current section. Function counters cannot be modified.

Elementary Function



With some elementary functions, the number of inputs can be increased.

Elementary Function Block

Elementary function blocks have internal states. If the input values are the same, the value on the output can be different each time the function is called. e.g. for a counter the value on the output is incremented.

An elementary function block is represented graphically as a frame with inputs and outputs. The inputs are always represented on the left and the outputs always on the right of the frame. The name of the function block, i.e. the function block type, is displayed in the center of the frame. The instance name is displayed above the frame.

Function blocks can have more than one output.

The name of the function block, i.e. the function block type, is displayed in the center of the frame.

The execution number, page 309 for the function block is shown to the right of the function block type.

The instance name is displayed above the frame.

The instance name serves as a unique identification for the function block in a project.

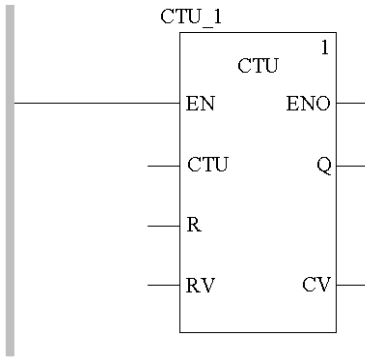
The instance name is created automatically and has the following structure: `TYPE_n` where `TYPE` is the function block type name: `TYPE_n`

- TYPE = Function block type name
- n = sequential number of the function block in the project

This automatically generated name can be modified for clarification. The instance name (max. 32 characters) must be unique throughout the project and is not case-sensitive. The instance name must conform to general naming conventions.

NOTE: To conform to IEC61131-3, only letters are permitted as the first character of the name. If you want to use a numeral as your first character however, this must be enabled explicitly.

Elementary Function Block

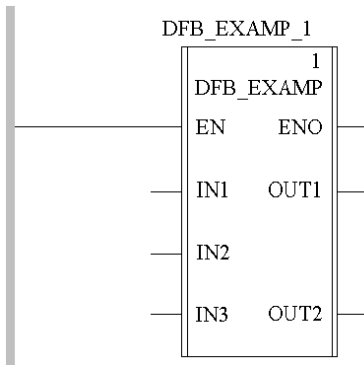


DFB

Derived function blocks (DFBs) have the same properties as elementary function blocks. The user can create them in the programming languages FBD, LD, IL, and/or ST.

The only difference to elementary function blocks is that the derived function block is represented as a frame with double vertical lines.

Derived Function Block



Procedure

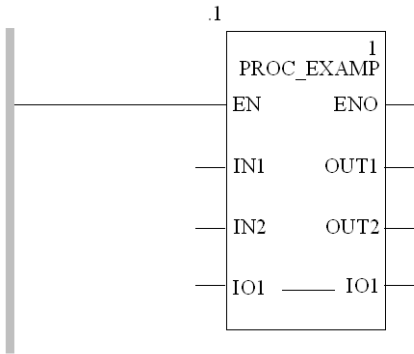
Procedures are functions viewed technically.

The only difference to elementary functions is that procedures can occupy more than one output and they support data type VAR_IN_OUT.

To the eye, procedures are no different than elementary functions.

Procedures are a supplement to IEC 61131-3 and must be enabled explicitly.

Procedure

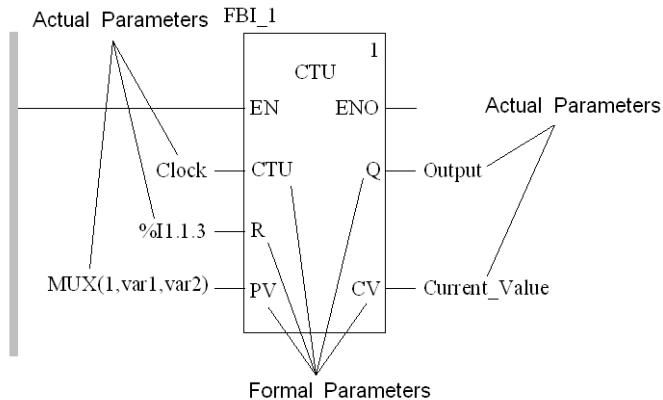


Parameters

Inputs and outputs are required to transfer values to or from an FFB. These are called formal parameters.

Objects are linked to formal parameters; these objects contain the current process states. They are called actual parameters.

Formal and actual parameters:



At program runtime, the values from the process are transferred to the FFB via the actual parameters and then output again after processing.

Only one object (actual parameter) of the following types may be linked to FFB inputs:

- Contact
- Variable
- Address
- Literal
- ST Expression

ST expressions on FFB inputs are a supplement to IEC 61131-3 and must be enabled explicitly.

- Link

The following combinations of objects (actual parameters) can be linked to FFB outputs:

- one or more coils
- one or more contacts
- one variable
- a variable and one or more connections (but not for VAR_IN_OUT, page 293 outputs)
- an address
- an address and one or more connections (but not for VAR_IN_OUT, page 293 outputs)
- one or more connections (but not for VAR_IN_OUT, page 293 outputs)

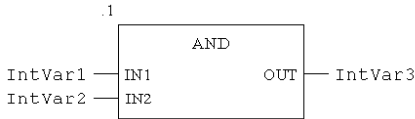
The data type of the object to be linked must be the same as that of the FFB input/output. If all actual parameters consist of literals, a suitable data type is selected for the function block.

Exception: For generic FFB inputs/outputs with data type ANY_BIT, it is possible to link objects of data type INT or DINT (not UINT and UDINT).

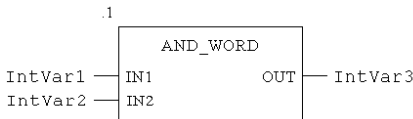
This is a supplement to IEC 61131-3 and must be enabled explicitly.

Example:

Allowed:



Not allowed:



(In this case, AND_INT must be used.)

Not all formal parameters have to be assigned an actual parameter. However, this does not apply in the case of negated pins. These must always be assigned an actual parameter. This is also the case with some formal parameter types. These types are shown in the following table.

Table of formal parameter types:

| Parameter type | EDT | STRING | ARRAY | ANY_ARRAY | IODDT | Device DDT | STRUCT | FB | ANY |
|------------------|-----|--------|-------|-----------|-------|------------|--------|----|-----|
| EFB: Input | - | - | - | - | / | / | - | / | - |
| DFB: Output | - | - | + | / | / | / | - | / | + |
| EFB: VAR_IN_OUT | + | + | + | + | + | / | + | / | + |
| DFB: Input | - | - | - | - | / | + | - | / | - |
| DFB: VAR_IN_OUT | + | + | + | + | + | + | + | / | + |
| EFB: Output | - | - | + | + | + | / | - | / | + |
| EF: Input | - | - | - | - | + | / | - | + | - |
| EF: VAR_IN_OUT | + | + | + | + | + | / | + | / | + |
| EF: Output | - | - | - | - | - | - | - | / | - |
| Procedure: Input | - | - | - | - | + | / | - | + | - |

| Parameter type | EDT | STRING | ARRAY | ANY_ARRAY | IODDT | Device DDT | STRUCT | FB | ANY |
|--|-----|--------|-------|-----------|-------|------------|--------|----|-----|
| Procedure: VAR_IN_OUT | + | + | + | + | + | / | + | / | + |
| Procedure: Output | - | - | - | - | - | / | - | / | + |
| + Actual parameter required | | | | | | | | | |
| - Actual parameter not required, it's the general rule, but there are exceptions for some FFBs, for instance when some parameters are used to characterize the information we want to be given by the FFB. | | | | | | | | | |
| / not applicable | | | | | | | | | |

FFBs that use actual parameters on the inputs that have not yet received any value assignment, work with the initial values of these actual parameters.

If no value is allocated to a formal parameter, then the initial value will be used for executing the function block. If no initial value has been defined then the default value (0) is used.

If a formal parameter is not assigned a value and the function block/DFB is instanced more than once, then the subsequent instances are run with the old value.

NOTE: An ANY_ARRAY_xxx input pin not connected will create automatically an hidden array of 1 element.

Public Variables

In addition to inputs/outputs, some function blocks also provide public variables.

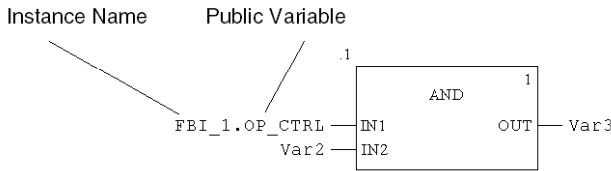
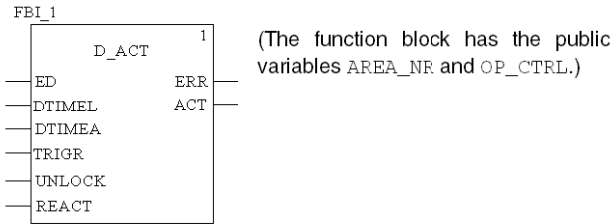
These variables transfer statistical values (values that are not influenced by the process) to the function block. They are used for setting parameters for the function block.

Public variables are a supplement to IEC 61131-3.

The assignment of values to public variables is made using their initial values.

Public variables are read via the instance name of the function block and the names of the public variables.

Example:



Private Variables

In addition to inputs, outputs and public variables, some function blocks also provide private variables.

Like public variables, private variables are used to transfer statistical values (values that are not influenced by the process) to the function block.

Private variables can not be accessed by user program. These type of variables can only be accessed by the animation table.

NOTE: Nested DFBs are declared as private variables of the parent DFB. So their variables are also not accessible through programming, but trough the animation table.

Private variables are a supplement to IEC 61131-3.

Programming Notes

Attention should be paid to the following programming notes:

- FFBs will only be processed when they are directly or indirectly connected to the left bus bar.
- If the FFB will be conditionally executed, the EN, page 291 input may be pre-linked through contacts or other FFBs.
- Boolean inputs and outputs can be inverted.
- Special conditions apply when using VAR_IN_OUT variables, page 293.
- Function block/DFB instances can be called multiple times, page 291.

Multiple Function Block Instance Call

Function block/DFB instances can be called more than once; other than instances from communication EFBs and function blocks/DFBs with an `ANY` output but no `ANY` input: these can only be called once.

Calling the same function block/DFB instance more than once makes sense, for example, in the following cases:

- If the function block/DFB has no internal value or it is not required for further processing.

In this case, memory is saved by calling the same function block/DFB instance more than once since the code for the function block/DFB is only loaded once.

The function block/DFB is then handled like a "Function".

- If the function block/DFB has an internal value and this is supposed to influence various program segments, for example, the value of a counter should be increased in different parts of the program.

In this case, calling the same function block/DFB means that temporary results do not have to be saved for further processing in another part of the program.

EN and ENO

One `EN` input and one `ENO` output can be used in all FFBs.

If the value of `EN` is equal to "0" when the FFB is invoked, the algorithms defined by the FFB are not executed and `ENO` is set to "0".

If the value of `EN` is equal to "1" when the FFB is invoked, the algorithms defined by the FFB will be executed. After the algorithms have been executed successfully, the value of `ENO` is set to "1". If an error occurs when executing these algorithms, `ENO` is set to "0".

If the `EN` pin is not assigned a value, when the FFB is invoked, the algorithm defined by the FFB is executed (same as if `EN` equals to "1"), Please refer to *Maintain output links on disabled EF* (see EcoStruxure™ Control Expert, Operating Modes).

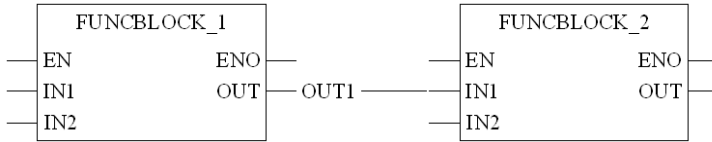
If `ENO` is set to "0" (caused by `EN=0` or an error during execution):

- Function blocks
 - `EN/ENO` handling with function blocks that (only) have one link as an output parameter:



If EN of FUNCBLOCK_1 is set to "0", the link on output OUT of FUNCBLOCK_1 maintains the old status it had during the last correctly executed cycle.

- EN/ENO handling with function blocks that have one variable and one link as output parameters:



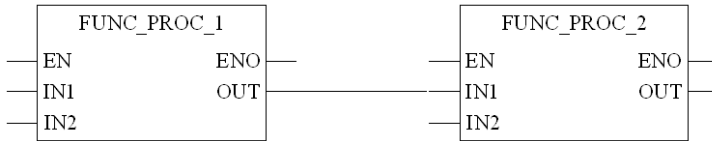
If EN of FUNCBLOCK_1 is set to "0", the link on output OUT of FUNCBLOCK_1 maintains the old status it had during the last correctly executed cycle. The OUT1 variable on the same pin either retains its previous status or can be changed externally without influencing the link. The variable and the link are saved independently of each other.

- Functions/Procedures

As defined in IEC61131-3, the outputs from deactivated functions (EN input set to "0") are undefined. (The same applies to procedures.)

Here nevertheless an explanation of the output statuses in this case:

- EN/ENO handling with function/procedure blocks that (only) have one link as an output parameter:



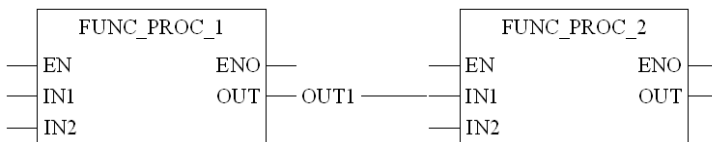
If EN of FUNC_PROC_1 is set to "0", the value of the link on output OUT of FUNC_PROC_1 depends on the project setting **Maintain output links on disabled EF**.

If this project setting is set to "0", the value of the link is set to "0".

If this project setting is set to "1", the link maintains the old value it had during the last correctly executed cycle.

For detailed information, please refer to *Maintain output links on disabled EF* (see EcoStruxure™ Control Expert, Operating Modes).

- EN/ENO handling with function/procedure blocks that have one variable and one link as output parameters:



If EN of FUNC_PROC_1 is set to "0", the value of the link on output OUT of FUNC_PROC_1 depends on the project setting **Maintain output links on disabled EF**.

If this project setting is set to "0", the value of the link is set to "0".

If this project setting is set to "1", the link maintains the old value it had during the last correctly executed cycle.

For detailed information, please refer to *Maintain output links on disabled EF* (see EcoStruxure™ Control Expert, Operating Modes).

The OUT1 variable on the same pin either retains its previous status or can be changed externally without influencing the link. The variable and the link are saved independently of each other.

The output behavior of the FFBs does not depend on whether the FFBs are invoked without EN/ENO or with EN=1.

NOTE: For disabled function blocks (EN = 0) with an internal time function (e.g. function block DELAY), time seems to keep running, since it is calculated with the help of a system clock and is therefore independent of the program cycle and the release of the block.

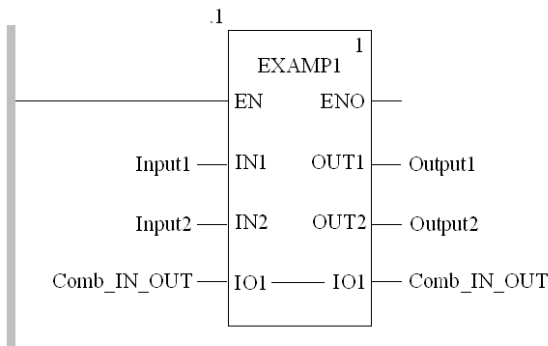
VAR_IN_OUT-Variable

FFBs are often used to read a variable at an input (input variables), to process it and to output the altered values of the **same** variable (output variables).

This special type of input/output variable is also called a VAR_IN_OUT variable.

The link between input and output variables is represented by a line in the FFB.

VAR_IN_OUT variable



The following special features are to be noted when using FFBs with VAR_IN_OUT variables.

- All VAR_IN_OUT inputs must be assigned a variable.

- Via graphical links only VAR_IN_OUT outputs with VAR_IN_OUT inputs can be connected.
- Only one graphical link can be connected to a VAR_IN_OUT input/output.
- A combination of variable/address and graphical connections is not possible for VAR_IN_OUT outputs.
- No literals or constants can be connected to VAR_IN_OUT inputs/outputs.
- No negations can be used on VAR_IN_OUT inputs/outputs.
- Different variables/variable components can be connected to the VAR_IN_OUT input and the VAR_IN_OUT output. In this case the value of the variables/variable component on the input is copied to the at the output variables/variable component.

Control Elements



Introduction

Control elements are used for executing jumps within an LD section and for returning from a subroutine (SRx) or derived function block (DFB) to the main program.

Control elements take up one cell.

Control Elements

The following control elements are available.

| Designation | Representation | Description |
|-------------|---|---|
| Jump |  | <p>When the status of the left link is 1, a jump is made to a label (in the current section).</p> <p>To generate an unconditional jump, the jump object must be placed directly on the left power rail.</p> <p>To generate a conditional jump, a jump object is placed at the end of a series of contacts.</p> |
| Label | LABEL : | <p>Labels (jump targets) are indicated as text with a colon at the end.</p> <p>This text is limited to 38 characters and must be unique within the entire section. The text must conform to general naming conventions.</p> <p>Jump labels can only be placed in the first cell directly on the power rail.</p> <p>Note: Jump labels may not "cut through" networks, i.e. an assumed line from the jump label to the right edge of the section may not be crossed by any object. This also applies to Boolean links and FFB links.</p> |
| Return |  | <p>RETURN objects cannot be used in the main program.</p> <ul style="list-style-type: none"> • In a DFB, a RETURN object forces the return to the program which called the DFB. <ul style="list-style-type: none"> ◦ The rest of the DFB section containing the RETURN object is not executed. ◦ The next sections of the DFB are not executed. <p>The program which called the DFB will be executed after return from the DFB.</p> <p>If the DFB is called by another DFB, the calling DFB will be executed after return.</p> • In an SR, a RETURN object forces the return to the program which called the SR. <ul style="list-style-type: none"> ◦ The rest of the SR containing the RETURN object is not executed. <p>The program which called the SR will be executed after return from the SR.</p> |


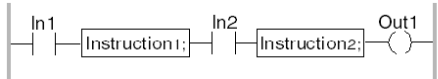

Operate Blocks and Compare Blocks

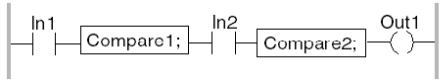
Introduction

In addition to the objects defined in IEC 61131-3, there are several other blocks for executing ST instructions, [page 421](#) and ST expressions, [page 420](#) and for simple compare operations. These blocks are only available in the LD programming language.

Objects

The following objects are available:

| Designation | Representation | Description |
|---------------------------|---|---|
| Operate block |  | <p>When the status of the left link is 1, the ST instruction in the block is executed.</p> <p>All ST instructions, page 421 are allowed except the control instructions:</p> <ul style="list-style-type: none"> • (RETURN, • JUMP, • IF, • CASE, • FOR • etc.) <p>For operate blocks, the state of the left link is passed to the right link (regardless of the result of the ST instruction).</p> <p>A block can contain up to 4096 characters. If not all characters can be displayed then the beginning of the character sequence will be followed by suspension points (...).</p> <p>An operate block takes up 1 line and 4 columns.</p> <p>Example:</p>  <p>In the example, <code>Instruction1</code> is executed if <code>In1=1</code>. <code>Instruction2</code> is executed if <code>In1=1</code> and <code>In2=1</code> (the result of <code>Instruction1</code> has no meaning for the execution of <code>Instruction2</code>). <code>Out1</code> becomes 1 if <code>In1=1</code> and <code>In2=1</code> (the results of <code>Instruction1</code> and <code>Instruction2</code> have no meaning for the status of <code>Out1</code>).</p> |
| Horizontal Matching Block |  | <p>Horizontal compare blocks used to execute a compare expression (<, >, <=, >=, =, <>) in the ST programming language. (Note: The same functionality is also possible using ST expressions, page 420.)</p> <p>A compare block performs an AND of its left In-pin and the result of its compare condition and assigns the result of this AND unconditionally to its right Out-pin.</p> <p>For example, if the state of the left link is 1 and the result of the comparison is 1, the state of the right link is 1.</p> <p>A horizontal matching block can contain up to 4096 characters. If not all characters can be displayed then the beginning of the character sequence will be followed by suspension points (...).</p> <p>A horizontal matching block takes up 1 line and 2 columns.</p> <p>Example:</p> |

| Designation | Representation | Description |
|-------------|----------------|---|
| | |  <p>In the example, Compare1 is executed if In1=1. Compare2 is executed if In1=1 , In2=1 a the result of Compare1=1. Out1 becomes 1 if In1=1, In2=1, the result of Compare1=1 and the result of Compare2=1.</p> |

Links

Description

Links are connections between LD objects (contacts, coils and FFBs etc.).

There are 2 different types of links:

- Boolean Links

Boolean links consist of one or more segments linking Boolean objects (contacts, coils) with one another.

There are different types of Boolean links as well:

- Horizontal Boolean Links

Horizontal Boolean links enable sequential contacts and coil switching.

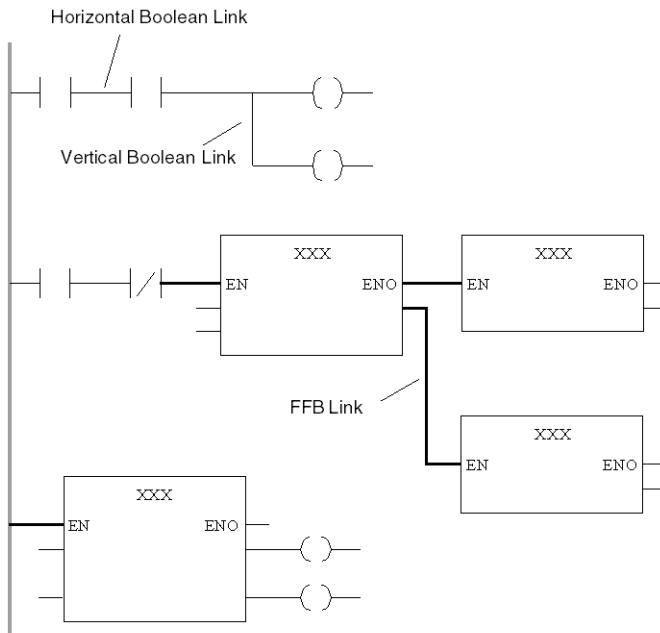
- Vertical Boolean Links

Vertical Boolean links enable parallel contacts and coil switching.

- FFB Links

FFB connections are a combination of horizontal and vertical segments that connect FFB inputs/outputs with other objects.

Connections:



General Programming Notes

Attention should be paid to the following general programming notes:

- The data types of the inputs/outputs to be linked must be the same.
- Links between parameters with variable lengths (e.g. `ANY_ARRAY_INT`) are not allowed.
- Several links can be connected with one output (right-hand side of one contact, one coil or one FFB output). However, only one link can be connected with an input (left-hand side of one contact, one coil or one FFB output).
- Unconnected contacts, coils and FFB inputs are specified as "0" by default.
- Links may not be used to create loops since the sequence of execution in this case cannot be clearly determined in the section. Loops must be created using actual parameters (see [Non-Permitted Loops](#), page 311).

Notes on Programming Boolean Links

Notes on Programming Boolean Links:

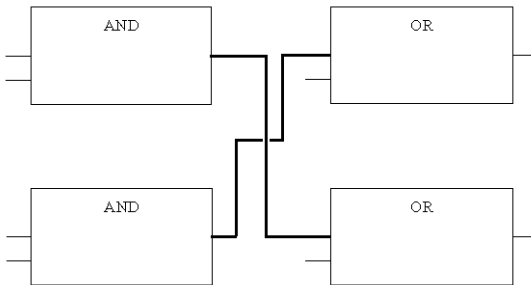
- Overlapping Boolean links with other objects is **not** permitted.

- The signal flow (power flow) is from left to right for Boolean links. Therefore, backwards links are not allowed.
- If two Boolean links are crossed, the links are connected automatically. Since crossing Boolean links is not possible, links are not indicated in any special way.

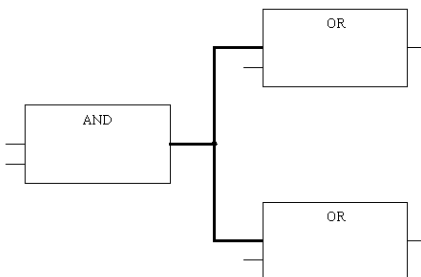
Notes on Programming FFB Links

Notes on Programming FFB Links:

- At least one side of an FFB link must be connected with an FFB input or output.
- To differentiate them from Boolean links, FFB links are shown with a doubly thick line.
- The signal flow (power flow) in FFB links is from the FFB output to the FFB input, no matter which direction they are made in. Therefore, backwards links are allowed.
- Only FFB inputs and FFB outputs may be linked to one-another. Linking more than one FFB outputs together is not possible. That means that no OR connection is possible in LD using FFB links.
- Overlapping FFB links with other objects is permitted.
- Crossing FFB links is also permitted. Crossed links are indicated by a "broken" link.



- Connection points between more FFB links are shown with a filled circle.

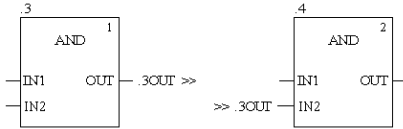


- To avoid links crossing each other, FFB links can also be represented in the form of connectors.

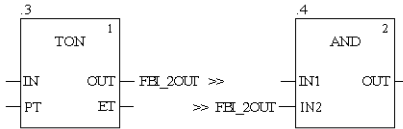
The source and target for the FFB connection are labeled with a name that is unique within the section.

The connector name has the following structure depending on the type of source object for the connection:

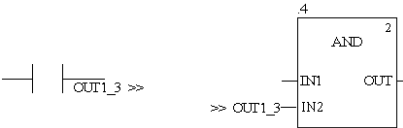
- For functions: "Function counter/formal parameter" for the source of the connection



- For function blocks: "Instance name/formal parameter" for the source of the connection



- For contacts: "OUT1_ sequential number"



Vertical Links

The "Vertical Link" is special. The vertical link serves as a logical OR. With this form of the OR link, 32 inputs (contacts) and 64 outputs (coils, links) are possible.

Text Object

Introduction

Text can be positioned as text objects in the Ladder Diagram (LD). The size of these text objects depends on the length of the text. The size of the object, depending on the size of the text, can be extended vertically and horizontally to fill further grid units. Text objects may overlap with other objects.

Edge Recognition

Introduction

During the edge recognition, a bit is monitored during a transition from 0 -> 1 (positive edge) or from 1 -> 0 (negative edge).

For this, the value of the bit in the previous cycle is compared to the value of the bit in the current cycle. In this case, not only the current value, but also the old value, are needed.

Instead of a bit, 2 bits are therefore needed for edge recognition (current value and old value).

Because the data type `BOOL` only offers one single bit (current value), there is another data type for edge recognition, `EBOOL` (expanded `BOOL`). In addition to edge recognition, the data type `EBOOL` provides an option for forcing. It must also be saved whether forcing the bit is enabled or not.

The data type `EBOOL` saves the following data:

- the current value of the bit in *Value bit*
- the old value of the bit in *History bit*
(the content of the value bit is copied to the History bit at the beginning of each cycle)
- Information whether forcing of the bit is enabled in *Force-Bit*
(0 = Forcing disabled, 1 = Forcing enabled)

Restrictions for EBOOL

CAUTION

UNINTENDED EQUIPMENT OPERATION

To perform a good edge detection the `%M` must be updated at each task cycle. When performing a unique writing, the edge will be infinite.

Failure to follow these instructions can result in injury or equipment damage.

Using an `EBOOL` variable for contacts to recognize positive (P) or negative (N) edges or with an EF called RE or FE, you have to adhere to the restrictions described below.

EBOOL with %M not written inside program

An `EBOOL` variable with a `%M` address, which is not written inside your program but directly, for example by an animation table, an operator screen or an HMI, will not work in the expected way. The edge is TRUE infinitely because the `%M` is only written one time.

NOTE: To avoid this issue the %M has to be written at the end of the task to update the old value information.

The old value is only updated, when the %M bit is written, so if you write the bit only one time, the edge detection will be infinite.

| Old Value | Current Value | Edge Detect | Description |
|-----------|---------------|-------------|--|
| 0 | 0 | 0 | state 0 (before writing the bit) |
| 0 | 1 | 1 | Write 1 in the bit (e.g. by animation table). |
| 0 | 1 | 1 | If you do not write again, the edge remains infinitely. |
| 1 | 1 | 0 | Write 1 again in the bit, the old value is updated and the edge detection is set to 0. |

EBOOL with %M written inside program

For an EBOOL variable with a %M address, which is written inside your program, you have to adhere to the restrictions described below:

- Do not use the bit with a SET or RESET coil. In this case the old value is not updated. So you can perform an infinite edge.
- Do not write the bit conditionally. A simple logic as `IF NOT %M1 THEN %M1 := TRUE; END_IF` leads to an infinite edge, because it is written only one time.

EBOOL with %I

For an EBOOL variable with a %I address you have to adhere to the restriction described below:

- When using multitasking the test of %I edge must be performed in the task where it is updated. The use of the edge detection of a %I scheduled in a task of higher priority must be avoided.

Example: If you have a fast task, which updates a %I, do not use a edge detection in the mast task. Depending on the scheduling you can detect the edge or not.

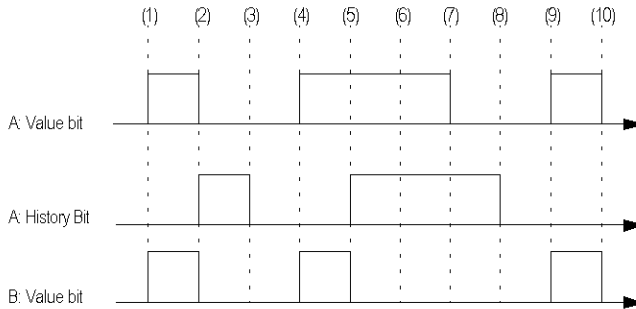
Recognizing Positive Edges

A contact to recognize positive edges is used to recognize positive edges. With this contact, the right connection for a program cycle is 1 when the transition of the associated actual parameter (A) is from 0 to 1 and, at the same time, the status of the left connection is 1. Otherwise, the status of the right link is 0.

In the example, a positive edge of the variable A is supposed to be recognize and B should therefore be set for a cycle.



Anytime the value bit of A equals 1 and the history bit equals 0, B is set to 1 for a cycle (cycle 1, 4, and 9).



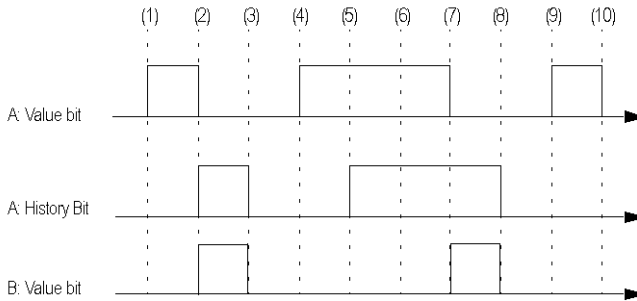
Recognizing Negative Edges

A contact to recognize negative edges is used to recognize negative edges. With this contact, the right connection for a program cycle is 1 when the transition of the associated actual parameter (A) is from 1 to 0 and, at the same time, the status of the left connection is 1. Otherwise, the status of the right link is 0.

In the example, a negative edge of the variable A is supposed to be recognize and B should therefore be set for a cycle.



Anytime the value bit of A equals 0 and the history bit equals 1, B is set to 1 for a cycle (cycle 2 and 8).



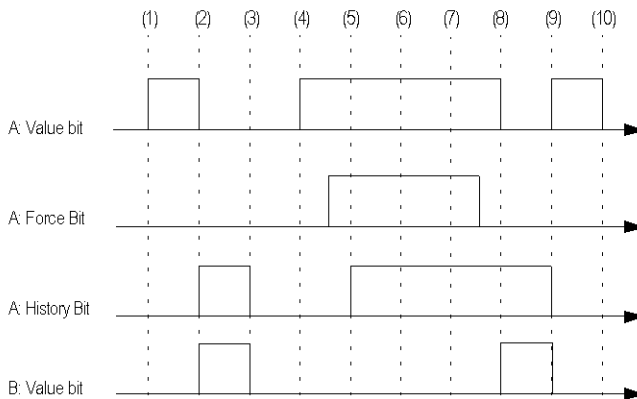
Forcing Bits

When forcing bits, the value of the variable determined by the logic will be overwritten by the force value.

In the example, a negative edge of the variable A is supposed to be recognized and B should therefore be set for a cycle.



Anytime the value bit or force bit of A equals 0 and the history bit equals 1, B is set to 1 for a cycle (cycle 1 and 8).



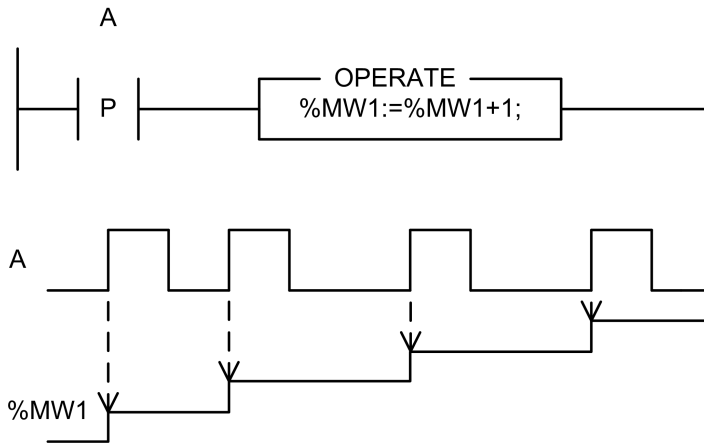
Using BOOL and EBOOL Variables

Edge recognition behavior using `BOOL` or `EBOOL` variables types can be different:

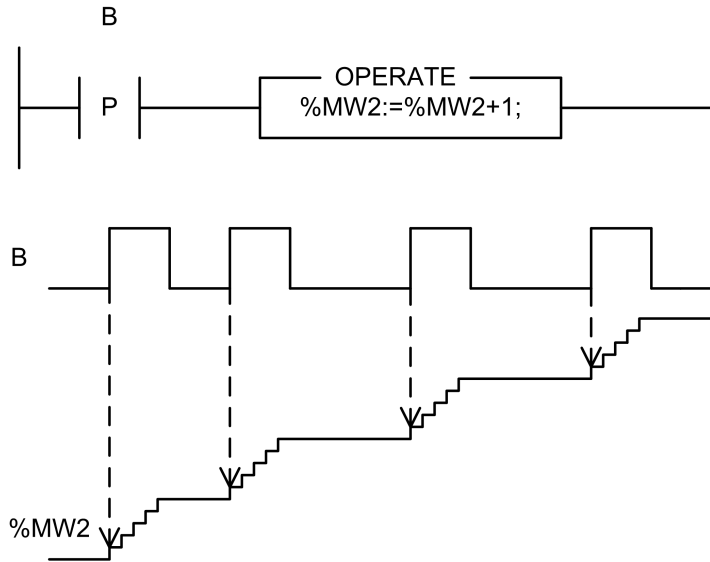
- When using a `BOOL` variable, the system manages the history by allowing edge detection during the contact execution.
- When using an `EBOOL` variable, the history bit is updated during the coil execution.

The following examples show the different behavior depending on the variable type.

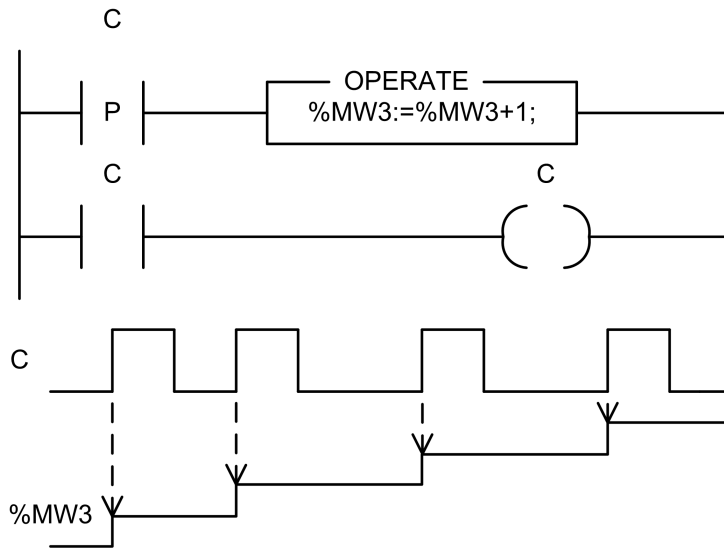
Variable `A` is define as `BOOL`, whenever `A` is set to 1, `%MW1` is incremented by 1.



Variable B is defined as EBOOL, the behavior is different when compared with variable A. While B is set to 1, %MW2 is incremented by 1 because the history bit is not updated.



Variable C is defined as EBOOL, the behavior is identical than variable A. The history bit is updated.

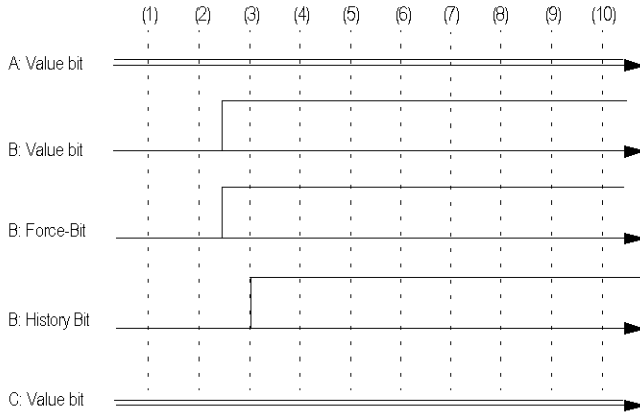
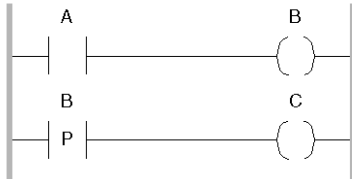


Forcing of Coils Can Cause the Loss of Edge Recognition

Forcing of coils can cause the loss of edge recognition.

In the example, when A equals 1, B should equal 1, and with a rising edge from A, the coil B will be set for a cycle.

In this example, the variable B is first assigned to the coil, and then to the link to recognize positive edges.



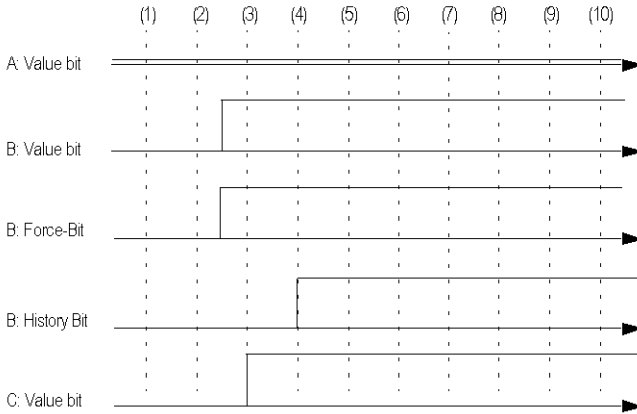
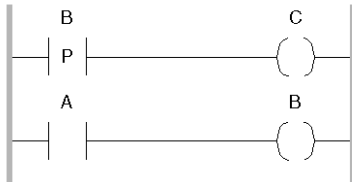
At the beginning of the second cycle, the value bit of B equals 0. When forcing B within this cycle, the force bit and value bit are set to 1. While processing the first line of the logic in the third cycle, the history bit of the coil (B) will also be set to 1.

Problem:

During edge recognition (comparison of the value bit and the history bit) in the second line of the logic, no edge is recognized, because due to the updating, the value bit and history bit on line 1 of B are always identical.

Solution:

In this example, the variable **B** is first assigned to the link to recognize positive edges and then the coil.



At the beginning of the second cycle, the value bit of **B** equals 0. When forcing **B** within this cycle, the force bit and value bit are set to 1. While processing the first line of the logic in the third cycle, the history bit of the link (**B**) will remain set to 0.

Edge recognition recognizes the difference between value bits and history bit and sets the coil (**C**) to 1 for one cycle.

Using Set Coil or Reset Coil Can Cause the Loss of Edge Recognition

Using set coil or reset coil can cause the loss of edge recognition with `EBOOL` variables.

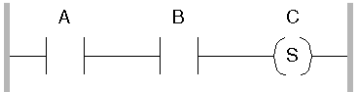
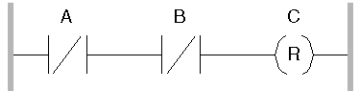

The variable above the set/reset coil (variable **C** in the example) is always affected by the value of the left link.

If the left link is 1, the value bit (variable **C** in the example) is copied to the history bit and the value bit is set to 1.

If the left link is 0, the value bit (variable **C** in the example) is copied to the history bit, but the value bit is not changed.

This means that whatever value the left link has before the set or reset coil, the history bit is always updated.

In the example, a positive edge of the variable C should be recognized and set D for a cycle.

| Code line | Behavior in LD | Corresponds to in ST |
|-----------|--|---|
| 1 | <p>Original situation: C = 0, History bit = 0</p>  <p>A = 1, B = 1, C = 1, History bit = 0</p> | <pre>IF A AND B THEN C := 1; ELSE C := C; END_IF;</pre> |
| 2 |  <p>A = 1, B = 1, C = 1, History = 1</p> | <pre>IF NOT (A) AND NOT (B) THEN C := 0; ELSE C := C; END_IF;</pre> |
| 3 |  <p>C = 1, History = 1</p> <p>D = 0, as the value bit and history bit of C are identical.</p> <p>The rising edge of C, shown in code line 1, is not recognized by the code in line 2, as this forces the history bit to be updated.</p> <p>(If the condition is FALSE, the present value of C is again assigned to C, see ELSE statement in code line 2 in ST example.)</p> | - |

Execution Sequence and Signal Flow

Execution Sequence of Networks

The following rules apply to network execution sequences:

- Executing a section is completed network by network based on the object links from above and below.

- Links may not be used to create loops since the sequence of execution in this case cannot be clearly determined. Loops must be created using actual parameters (see Loop Planning, page 311).
- The execution sequence of networks which are only linked by the left power rail, is determined by the graphical sequence (from top to bottom) in which these are connected to the left power rail. This does not apply if the sequence is influenced by control elements.
- Processing on a network is ended completely before the processing begins on another network.
- No element of a network is deemed to be processed until the status of all inputs of this element have been processed.
- Processing on a network is only ended if all outputs on this network have been processed. This also applies if the network contains one or more control elements.

Signal Flow within a Network

For signal flow within a network (rungs), the following rules apply:

- The signal flow for Boolean links is:
 - left to right with horizontal Boolean links and
 - from top to bottom with vertical Boolean links.
- The signal flow with FFB links is from the FFB output to the FFB input, regardless of which direction they are made in.
- An FFB is only processed if all elements (FFB outputs etc.) to which it's inputs are linked are processed.
- The execution sequence of FFBs that are linked with various outputs of the same FFB runs from top to bottom.
- The execution sequence of objects is not influenced by their positions within the network.
- The execution sequence for FFBs is represented as execution number by the FFB.

Priorities

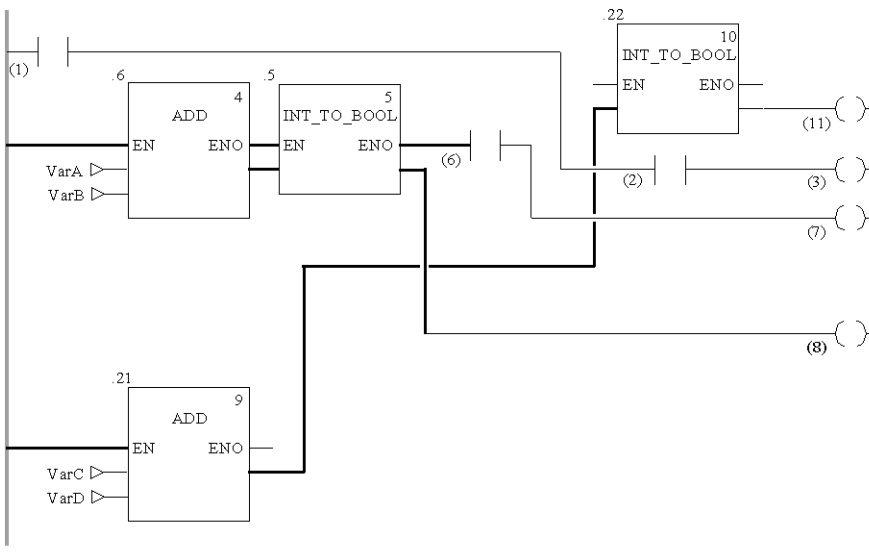
Priorities when defining the signal flow within a section:

| Priority | Rule | Description |
|----------|--------------------|--|
| 1 | Link | Links have the highest priorities in defining the signal flow within an LD section. |
| 2 | Network by Network | Processing on a network is ended completely before the processing begins on another network. |

| Priority | Rule | Description |
|----------|-----------------|--|
| 3 | Output sequence | Outputs of the same function block or outputs to vertical links are processed from top to bottom. |
| 4 | Rung by Rung | Lowest priority. The execution sequence of networks which are only linked by the left power rail, is determined by the graphical sequence (from top to bottom) in which these are connected to the left power rail. (Only applies if none of the other rules apply). |

Example

Example of the execution sequence of objects in an LD section:



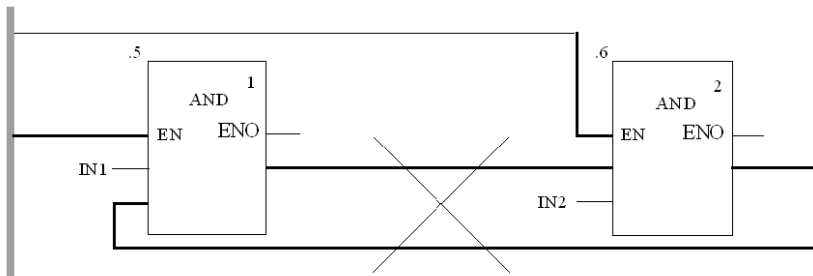
NOTE: The execution numbers for contacts and coils is not shown. They are only shown in the graphic to provide a better overview.

Loop Planning

Non-Permitted Loops

Creating loops using links alone is not permitted because it is not possible to clearly define the signal flow (the output of one FFB is the input of the next FFB, and the output of this one is the input of the first again).

Non-permitted loops via links:



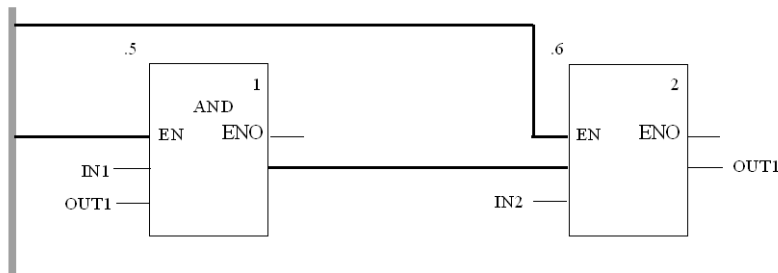
Generating Via an Actual Parameter

This type of logic must be generated using feedback variables so that the signal flow can be determined.

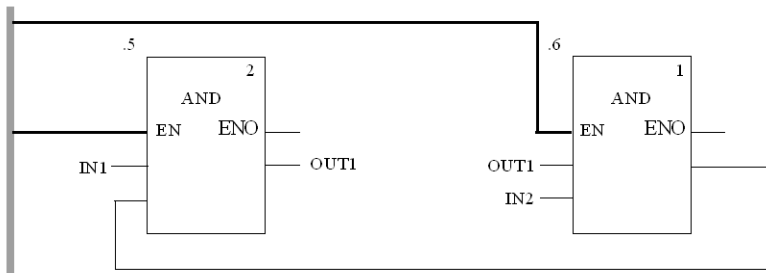
Feedback variables must be initialized. The initial value is used during the first execution of the logic. Once they have been executed the initial value is replaced by the actual value.

Pay attention to the two different types of execution sequences (number in brackets after the instance name) for the two blocks.

Loop generated with an actual parameter: Type 1



Loop generated with an actual parameter: Type 2



Change Execution Sequence

Introduction

The order of execution in networks and the execution order of objects within a network are defined by a number of rules, page 309.

In some cases the execution order suggested by the system should be changed.

The procedure for defining/changing the execution sequence of networks is as follows:

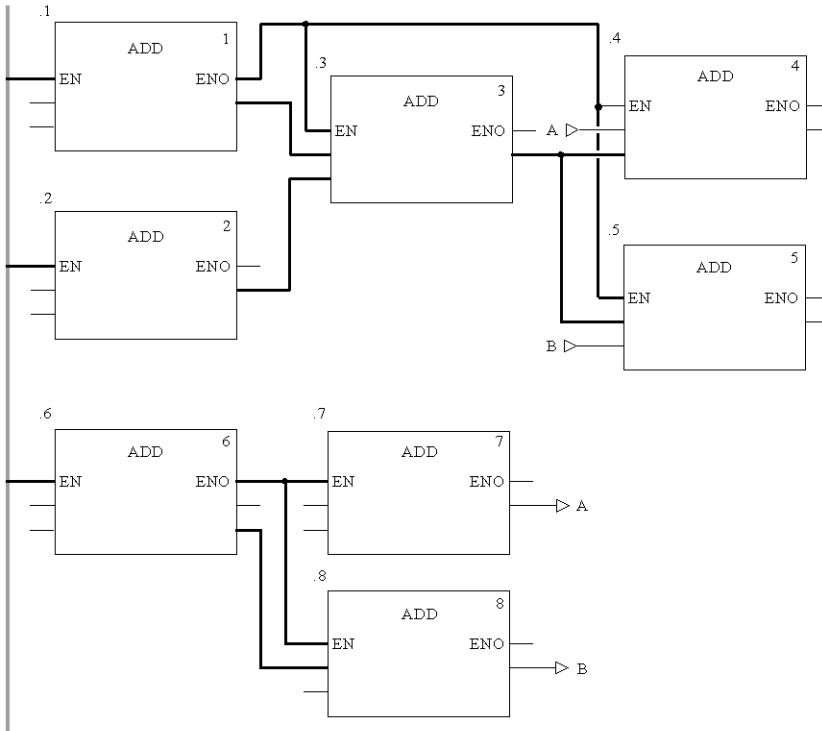
- Using Links Instead of Actual Parameters
- Network Positions

The procedure for defining/changing the execution sequence of networks is as follows:

- Positioning of Objects

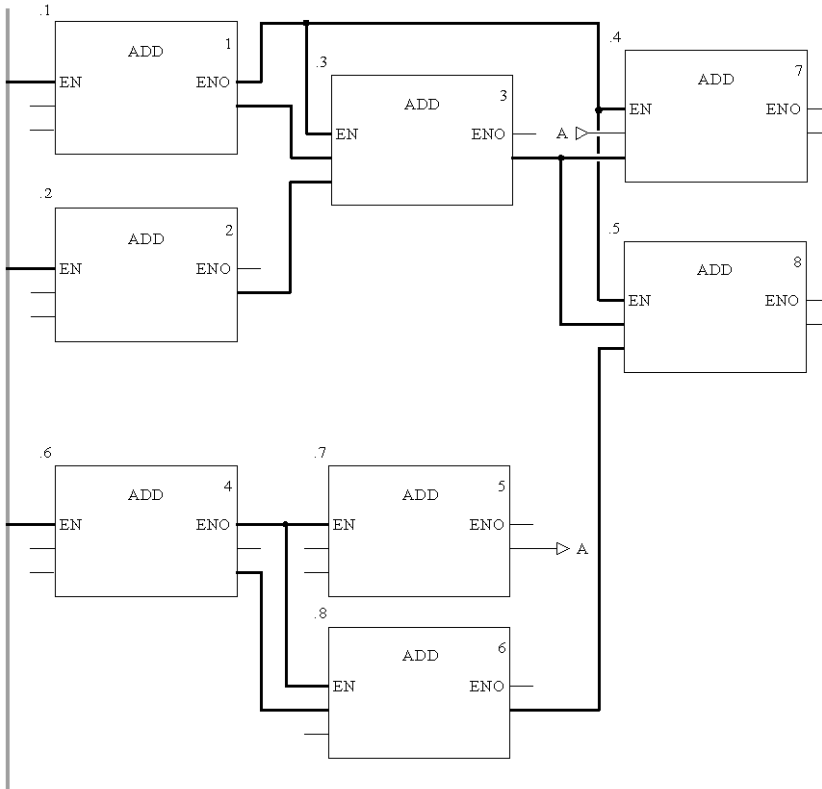
Original Situation

The following representation shows two networks for which the execution sequences are only defined by their position within the section, without taking into account that block 0 . 4 / 0 . 5 and 0 . 7 / 0 . 8 require another execution sequence.



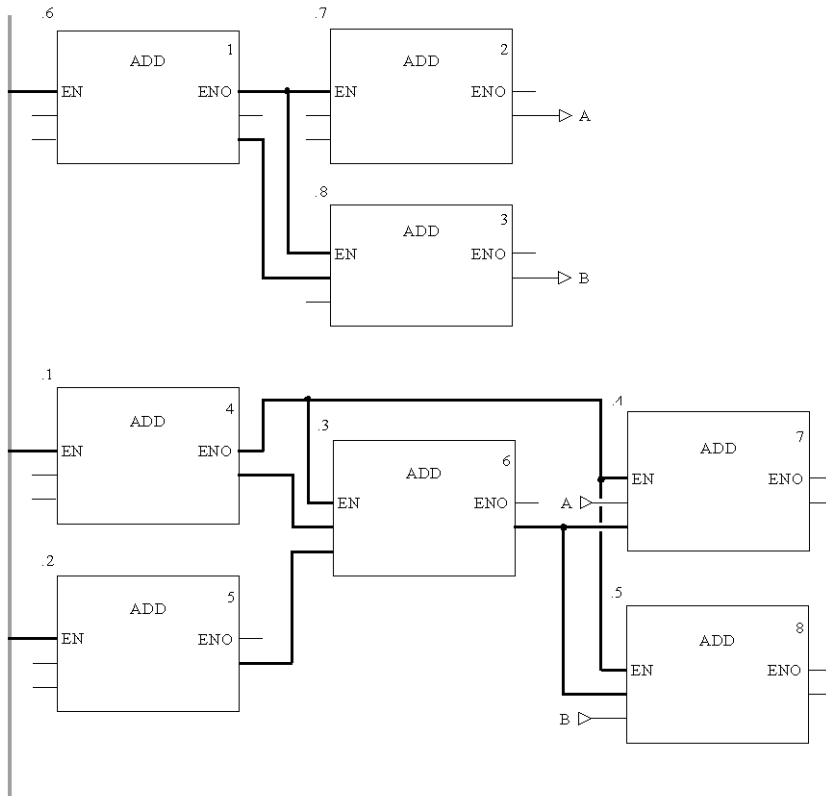
Link Instead of Actual Parameter

By using a link instead of a variable the two networks are run in the proper sequence (see also Original Situation, page 314).



Network Positions

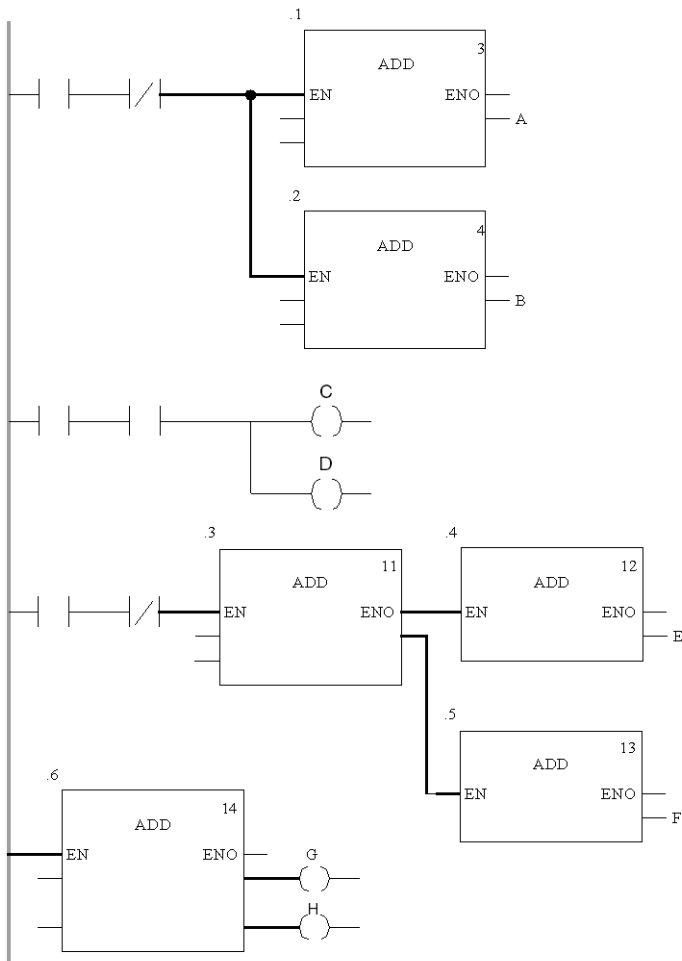
The correct execution sequence can be achieved by changing the position of the networks in the section (see also Original Situation, page 314).



Positioning of Objects

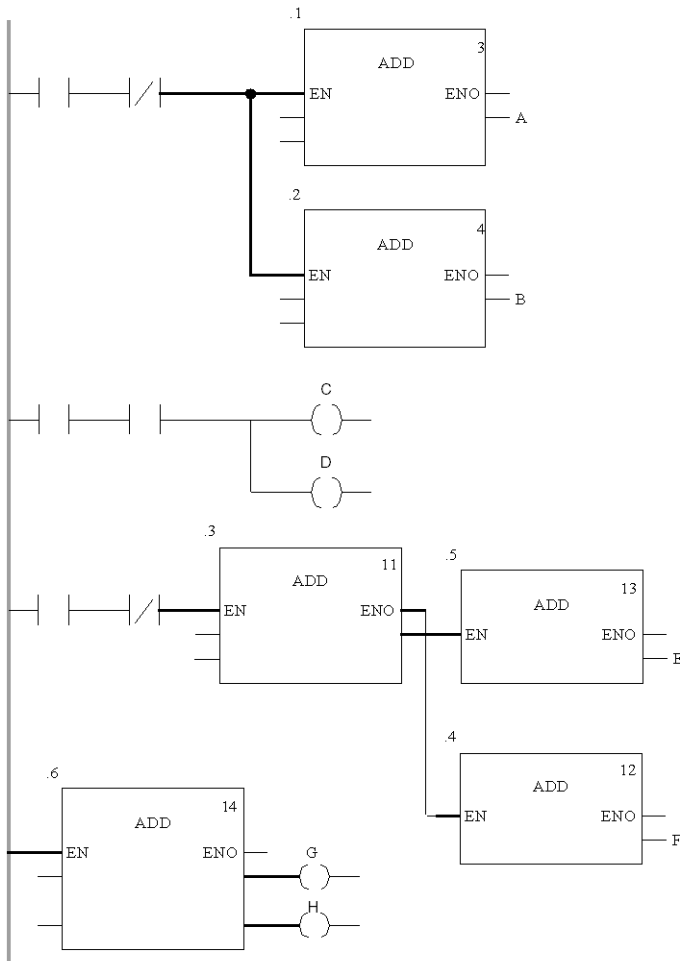
The position of objects can only have an influence on the execution order if several inputs (left link of Contacts/Coils, FFB inputs) are linked with the same output of the object "to be called" (right link of Contacts/Coils, FFB outputs) (see also Original Situation, page 314).

Original situation:



In the first network, block positions 0.1 and 0.2 are switched. In this case (common origins for both block inputs) the execution sequence of both blocks is switched as well (processed from top to bottom). The same applies when switching coils C and D in the second network.

In the third network, block positions 0.4 and 0.5 are switched. In this case (different origins for the block inputs) the execution sequence of the blocks is not switched (processed in the sequence that the block outputs are called in). The same applies when switching coils G and H in the last network.



SFC Sequence Language

What's in This Chapter

| | |
|---|-----|
| General Information about SFC Sequence Language | 319 |
| Steps and Macro Steps | 324 |
| Actions and Action Sections..... | 332 |
| Transitions and Transition Sections | 342 |
| Jump | 345 |
| Link | 346 |
| Branches and Merges | 347 |
| Text Objects | 350 |
| Single-Token | 350 |
| Multi-Token..... | 361 |

Overview

This chapter describes the SFC sequence language which conforms to IEC 61131-1.

General Information about SFC Sequence Language

Overview

This section contains a general overview of the SFC sequence language.

General Information about SFC Sequence Language

Introduction

The sequence language SFC (Sequential Function Chart), which conforms to IEC 61131-3, is described in this section.

Structure of a Sequence Controller

IEC conforming sequential control is created in Control Expert from SFC sections (top level), transition sections and action sections.

These SFC sections are only allowed in the Master Task of the project. SFC sections cannot be used in other tasks or DFBS.

In Single Token, each SFC section contains exactly one SFC network (sequence).

In Multi-Token, an SFC section can contain one or more independent SFC networks.

Objects

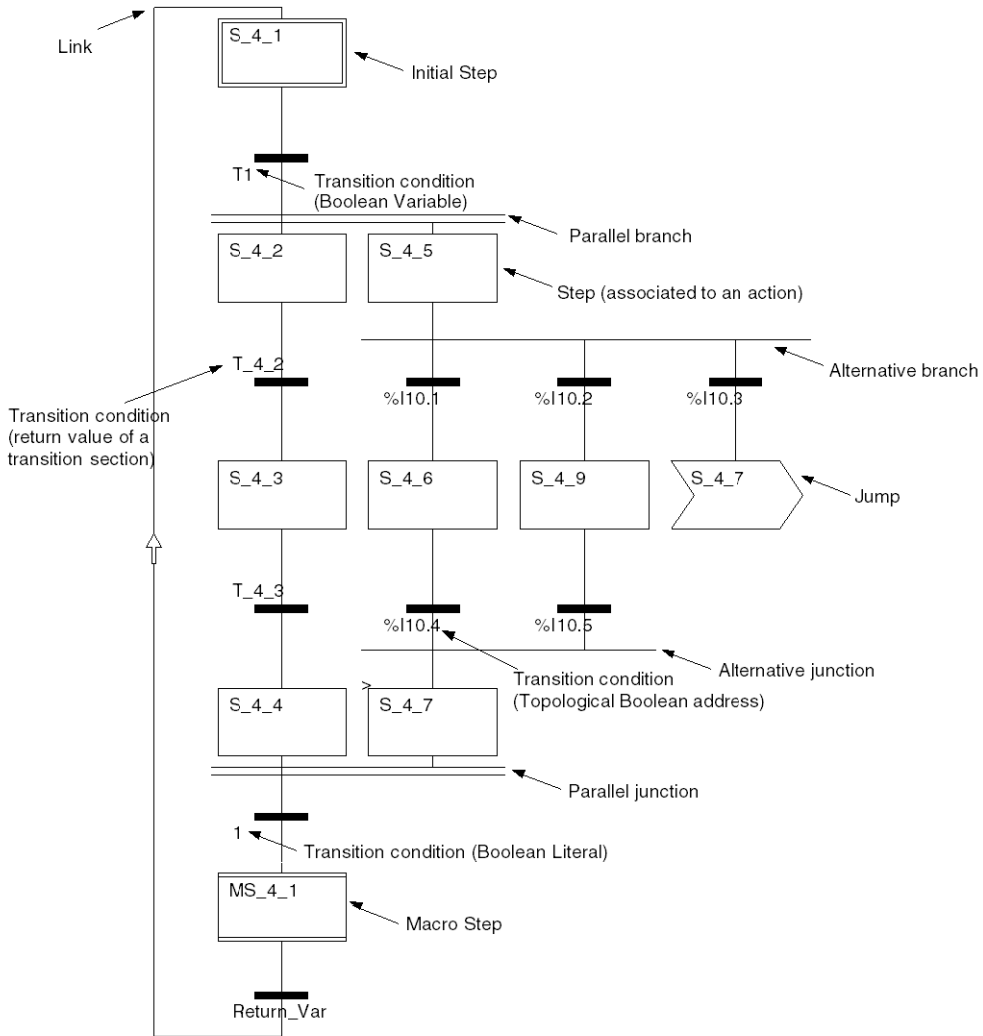
An SFC section provides the following objects for creating a program:

- Step, page 324
- Macro Step (embedded sub-step), page 327
- Transition (transition condition), page 342
- Jump, page 345
- Link, page 346
- Alternative branch, page 347
- Alternative junction, page 347
- Parallel branch, page 348
- Parallel junction, page 348

Comments regarding the section logic can be provided using text objects (related topics Text Object, page 350).

Representation of an SFC Section

Appearance:



Structure of an SFC Section

An SFC section is a "Status Machine", i.e. the status is created by the active step and the transitions pass on the switch/change behavior. Steps and transitions are linked to one another through directional links. Two steps can never be directly linked and must always be separated by a transition. The active signal status processes take place along the directional

links and are triggered by switching a transition. The direction of the chain process follows the directional links and runs from the end of the preceding step to the top of the next step. Branches are processed from left to right.

Every step has zero or more actions. A transition condition is necessary for every transition.

The last transition in the chain is always connected to another step in the chain (via a graphic link or jump symbol) to create a closed loop. Step chains are therefore processed cyclically.

SFCCHART_STATE Variable

When an SFC section is created, it is automatically assigned a variable of data type `SFCCHART_STATE`. The variable that is created always has the name of the respective SFC section.

This variable is used to assign the SFC control blocks to the SFC section to be controlled.

Token Rule

The behavior of an SFC network is greatly affected by the number of tokens selected, i.e. the number of active steps.

Explicit behavior is possible by using one token (single token). (Parallel branches each with an active token [step] per branch as a single token). This corresponds to a step chain as defined in IEC 61131-3).

A step chain with a number of maximum active steps (Multi Token) defined by the user increases the degree of freedom. This reduces/eliminates the restrictions for enforcing unambiguousness and non-blocking and must be guaranteed by the user. Step chains with Multi Token do not conform to IEC 61131-3.

Section Size

- An SFC section consists of a single-page window.
- Because of performance reasons, it is strongly recommended to create less than 100 SFC sections in a project (macro sections are not counted).
- The window has a logical grid of 200 lines and 32 columns.
- Steps, transitions and jumps each require a cell.
- Branches and links do not require their own cells, they are inserted in the respective step or transition cell.
- A maximum of 1024 steps can be placed per SFC section (including all their macro sections).

- A maximum of 100 steps can be active (Multi Token) per SFC section (including all their macro sections) .
- A maximum of 64 steps can be set manually at the same time per SFC section (Multi Token).
- A maximum of 20 actions can be assigned to each SFC step.
- The nesting depth of macros, i.e. macro steps within macro steps, is to 8 levels.

IEC Conformity

For a description of the extent to which the SFC programming language conforms to IEC, see IEC Conformity, page 508.

Link Rules

Link Rules

The table indicates which object outputs can be linked with which object inputs.

| From object output of | To object input of |
|-----------------------|---|
| Step | Transition |
| | Alternative Branch |
| | Parallel joint |
| Transition | Step |
| | Jump |
| | Parallel Branch |
| | Alternative joint |
| Alternative Branch | Transition |
| Alternative joint | Step |
| | Jump |
| | Parallel Branch |
| | Alternative joint |
| Parallel Branch | Step |
| | Jump |
| | Alternative joint (only with Multi-Token, page 361) |
| Parallel joint | Transition |

| From object output of | To object input of |
|-----------------------|---|
| | Alternative branch (only with Multitoken, page 361) |
| | Alternative joint |

Steps and Macro Steps

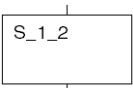
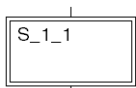
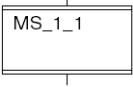
Overview

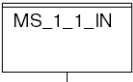
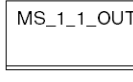
This section describes the step and macro step objects of the SFC sequence language.

Step

Step Types

The following types of steps exist:

| Type | Representation | Description |
|---------------|---|--|
| "Normal" Step |  | <p>A step becomes active when the previous step becomes inactive (a delay that may be defined must pass) and the upstream transition is satisfied. A step normally becomes inactive when a delay that may be defined passes and the downstream transition is satisfied. For a parallel joint, all previous steps must satisfy these conditions.</p> <p>Zero or more actions belong to every step. Steps without action are known as waiting steps.</p> |
| Initial step |  | <p>The initial status of a sequence string is characterized by the initial step. After initializing the project or initializing the sequence string, the initial step is active.</p> <p>Initial steps are not normally assigned with any actions.</p> <p>With Single-Token (Conforming with IEC 61131-3) only one initial step is allowed per sequence.</p> <p>With Multi-Token, a definable number (0 to 100) of initial steps are possible.</p> |
| Macro Step |  | See Macro Step, page 327 |

| Type | Representation | Description |
|-------------|---|---------------------------|
| Input step |  | see Input Step, page 328 |
| Output step |  | see Output Step, page 328 |

Step Names

When creating a step, it is assigned with a suggested number. The suggested number is structured as follows S_{i_j} , whereas i is the (internal) current number of the section and j is the (internal) current step number in the current section.

You can change the suggested numbers to give you a better overview. Step names (maximum 32 characters) must be unique over the entire project, i.e. no other step, variable or section etc. may exist with the same name. There are no case distinctions. The step name must correspond with the standardized name conventions.

Step Times

Each step can be assigned a minimum supervision time, a maximum supervision time and a delay time:

- **Minimum Supervision Time**

The minimum supervision time sets the minimum time for which the step should normally be active. If the step becomes inactive before this time has elapsed, an error message is generated. In animation mode, the error is additionally identified by a colored outline (yellow) around the step object.

If no minimum supervision time or a minimum supervision time of 0 is entered, step supervision is not carried out.

The error status remains the same until the step becomes active again.

- **Maximum Supervision Time**

The maximum supervision time specifies the maximum time in which the step should normally be active. If the step is still active after this time has elapsed, an error message is generated. In animation mode, the error is additionally identified by a colored outline (pink) around the step object.

If no maximum supervision time or a maximum supervision time of 0 is entered, step supervision is not carried out.

The error status remains the same until the step becomes inactive.

- **Delay Time**

The delay time (step dwell time) sets the minimum time for which the step must be active.

NOTE: The defined times apply for the step only, not for the allocated actions. Individual times can be defined for these.

Setting the Step Times

The following formula is to be used for defining/determining these times:

Delay time < minimum supervision time < maximum supervision time

There are 2 ways to assign the defined values to a step:

- As a duration literal
- Use of the data structure `SFCSTEP_TIMES`

SFCSTEP_TIMES Variable

Every step can be implicitly allocated a variable of data type `SFCSTEP_TIMES`. The elements for this data structure can be read from and written to (read/write).

The data structure is handled the same as any other data structure, i.e. they can be used in variable declarations and therefore accessing the entire data structure (e.g. as FFB parameter) is possible.

Structure of the Data Structure:

| Element Name | Data type | Description |
|-----------------|-----------|--------------------------|
| "VarName".delay | TIME | Delay Time |
| "VarName".min | TIME | Minimum Supervision Time |
| "VarName".max | TIME | Maximum Supervision Time |

SFCSTEP_STATE Variable

Every step is implicitly allocated a variable of data type `SFCSTEP_STATE`. This step variable has the name of the allocated step. The elements for this data structure can only be read (read only).

You can see the `SFCSTEP_STATE` variables in the **Data Editor**. The **Comment** for a `SFCSTEP_STATE` variable is the comment entered as a property of the step itself. Please refer to chapter Defining the properties of steps (see EcoStruxure™ Control Expert, Operating Modes).

The data structure cannot be used in variable declarations. Therefore, accessing the entire data structure (e.g. as FFB parameter) is not possible.

Structure of the Data Structure:

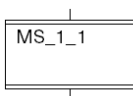
| Element Name | Data type | Description |
|--------------------|-----------|--|
| "StepName".t | TIME | Current dwell time in the step. If the step is deactivated, the value of this element is retained until the step is activated again. |
| "StepName".x | BOOL | 1: Step active 0: Step inactive |
| "StepName".tminErr | BOOL | This element is a supplement to IEC 61131-3. 1: Underflow of minimum supervision time 0: No underflow of minimum supervision time The element is automatically reset in the following cases: <ul style="list-style-type: none"> • If the step is activated again • If the sequence control is reset • If the command button Reset Time Error is activated |
| "StepName".tmaxErr | BOOL | This element is a supplement to IEC 61131-3. 1: Overflow of maximum supervision time 0: No overflow of maximum supervision time The element is automatically reset in the following cases: <ul style="list-style-type: none"> • If the step is exited • If the sequence control is reset • If the command button Reset Time Error is activated |

Macro Steps and Macro Sections

Macro Step

Macro steps are used for calling macro sections and thus for hierarchical structuring of sequential controls.

Representation of a Macro Step:



Macro steps have the following properties:

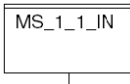
- Macro steps can be positioned in "Sequence Control" sections and in macro sections.
- The number of macro steps is unlimited.
- The nesting depth, i.e. macro steps within macro steps is to 8 levels.
- Each macro step is implicitly allocated a variable of data type `SFCSTEP_STATE`, see `SFCSTEP_STATE` Variable, page 326.
- Macro steps can be allocated a variable of data type `SFCSTEP_TIMES`, see `SFCSTEP_TIMES` Variable, page 326.
- Macro steps can NOT be allocated with actions.
- Each macro step can be replaced with the sequence string in the allocated macro section.

Macro steps are a supplement to IEC 61131-3 and must be enabled explicitly.

Input Step

Every macro section begins with an input step.

Representation of an input step:



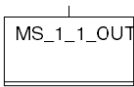
Input steps have the following properties:

- Input steps are automatically placed in macro sections by the SFC editor.
- Only 1 individual input step is placed for each macro section.
- An input step cannot be deleted, copied or inserted manually.
- Each input step is implicitly allocated a variable of data type `SFCSTEP_STATE`, see `SFCSTEP_STATE` Variable, page 326.
- Input steps can be allocated a variable of data type `SFCSTEP_TIMES`, see `SFCSTEP_TIMES` Variable, page 326.
- Input steps can be allocated actions.

Output Step

Every macro section ends with an output step.

Representation of an output step:



Output steps have the following properties:

- Output steps are automatically placed in macro sections by the SFC editor.
- Only 1 individual output step is placed for each macro section.
- An output step cannot be deleted, copied or inserted manually.
- Output steps can NOT be allocated with actions.
- Output steps can only be assigned a delay time. Assigning supervision times is not possible, see [Step Times](#), page 325.

Macro Section

A macro section consists of a single sequence string having principally the same elements as a "sequence control" section (e.g. steps, initial step[s], macro steps, transitions, branches, joints, etc.).

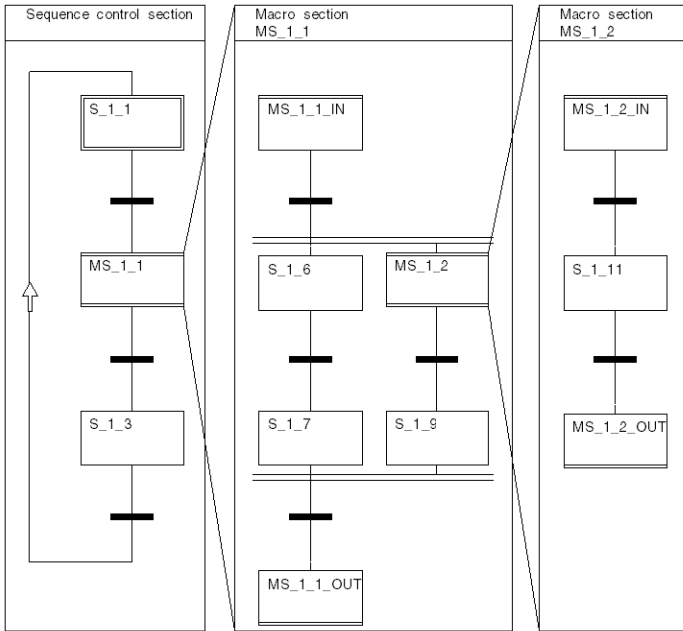
Additionally, each macro section contains an input step at the beginning and an output step at the end.

Each macro step can be replaced with the sequence string in the allocated macro section.

Therefore, macro sections can contain 0, 1 or more initial steps, see also [Step Types](#), page 324.

- Single-Token
 - 0 Initial steps
are used in macro sections, if there is already an initial step in the higher or lower section.
 - 1 Initial step
is used in macro sections, if there are no initial steps in the higher or lower section.
- Multi-Token
A maximum of 100 initial steps can be placed per section (including all their macro sections).

Using macro sections:



The name of the macro section is identical to the name of the macro step that it is called from. If the name of the macro step is changed then the name of the respective macro section is changed automatically.

A macro section can only be used once.

Macro Step Processing

Macro Step Processing:

| Phase | Description |
|-------|--|
| 1 | A macro step is activated if the previous transition condition is TRUE. At the same time, the input step in the macro section is activated. |
| 2 | The sequence string of the macro section is processed. The macro step remains active as long as at least one step in the macro section is active. |
| 3 | If the output step of the macro section is active then the transitions following the macro step are enabled. |
| 4 | The macro step becomes inactive when the output step is activated which causes the following transition conditions to be enabled and the transition condition to be TRUE. At the same time, the output step in the macro section is activated. |

Step Names

When creating a step, it is assigned with a suggested number.

Meanings of the Suggested Numbers:

| Step Type | Suggested Number | Description |
|---|------------------|---|
| Macro Step | MS_i_j | MS = Macro Step i = (internal) current (sequential) number of the current section j = (internal) current (sequential) macro step number of the current section |
| Input step | MS_k_l_IN | MS = Macro Step k = (internal) current (sequential) number of the calling section l = (internal) current (sequential) macro step number of the calling section IN = Input Step |
| Output step | MS_k_l_OUT | MS = Macro Step k = (internal) current (sequential) number of the calling section l = (internal) current (sequential) macro step number of the calling section OUT = Output Step |
| "Normal" Step (within a macro section) | S_k_m | S = Step k = (internal) current (sequential) number of the calling section m = (internal) current (sequential) step number of the calling section |

You can change the suggested numbers to give you a better overview. Step names (maximum 28 characters for macro step names, maximum 32 characters for step names) must be unique within the entire project, i.e. no other step, variable or section (with the exception of the name of the macro section assigned to the macro step) etc. may exist with the same name. There are no case distinctions. The step name must correspond with the standardized name conventions.

If the name of the macro step is changed then the name of the respective macro section and the steps within it are changed automatically.

For example If MS_1_1 is renamed to MyStep then the step names in the macro section are renamed to MyStep_IN, MyStep_1, ..., MyStep_n, MyStep_OUT.

Actions and Action Sections

Overview

This section describes the actions and action sections of the SFC sequence language.

Action

Introduction

Actions have the following properties:

- An action can be a Boolean variable (action variable, page 332) or a section (action section, page 333) of programming language FBD, LD, IL or ST.
- A step can be assigned none or several actions. A step which is assigned no action has a waiting function, i.e. it waits until the assigned transition is completed.
- If more than one action is assigned to a step they are processed in the sequence in which they are positioned in the action list field.

Exception: Independent of their position in the action list field, actions with the qualifier P1 are always processed first and actions with the qualifier P0 are processed last.

- The control of actions is expressed through the use of qualifiers, page 335.
- A maximum of 20 actions can be assigned to each step.
- The action variable that is assigned to an action can also be used in actions from other steps.
- The action variable can also be used for reading or writing in any other section of the project (multiple assignment).
- Actions that are assigned an qualifier with duration can only be activated one time.
- Only Boolean variables/addresses or Boolean elements of multi-element variables are allowed as action variables.
- Actions have unique names.

The name of the action is either the name of the action variable or the name of the action section.

Action Variable

The following are authorized as action variables:

- Address of data type `BOOL`

An action can be assigned to a hardware output using an address. In this case, the action can be used as enable signal for a transition, as input signal in another section and as output signal for the hardware.

- Simple variable or element of a multi-element variable of data type `BOOL`

The action can be used as an input signal with assistance from a variable in another section.

- Unlocated Variable

With unlocated variables, the action can be used as enable signal for a transition and as input signal in another section.

- Located Variable

With located variables the action can be used as an enabling signal for a transition, as an input signal in another section and as an output signal for the hardware.

Action Names

If an address or a variable is used as an action then that name (e.g. %Q10.4, Variable1) is used as the action name.

If an action section is used as an action then the section name is used as the action name.

Action names (maximum 32 characters) must be unique over the entire project, i.e. no other transition, variable or section etc. may exist with the same name. There are no case distinctions. The action name must correspond with the standardized name conventions.

Action Section

Introduction

An action section can be created for every action. This is a section which contains the logic of the action and it is automatically linked with the action.

Name of the Action Section

The name of the action section is always identical to the assigned action, page 333.

Programming Languages

FBD, LD, IL and ST are possible as programming languages for action sections.

Properties of Action Sections

Action sections have the following properties:

- Action sections can have any amount of outputs.
- Subroutine calls are only possible in action sections when Multitoken operation is enabled.
 - Note:** The called subroutines are **not** affected by the controller of the sequence string, i.e.
 - the qualifier assigned to the called action section does not affect the subroutine
 - the subroutine also remains active when the called step is deactivated
- No diagnosis functions, diagnosis function blocks or diagnosis procedures may be used in action sections.
- Action sections can have any amount of networks.
- Action sections belong to the SFC section in which they were defined and can be assigned any number of actions within this SFC section (including all of their macro sections).
- Action sections which are assigned an qualifier with duration, can only be activated one time.
- Action sections belong to the SFC section that they were defined in. If the respective SFC section is deleted then all action sections of this SFC section are also deleted automatically.
- Action sections can be called exclusively from actions.

SFCACTION_STATE Variable

When final scan logic is activated, every action section is implicitly allocated a variable of data type SFCACTION_STATE. This variable has the name of the allocated action section. The elements for this data structure are read only.

You can see the SFCACTION_STATE variables in the **Data Editor**. The **Comment** for a SFCACTION_STATE variable is the comment entered as a property of the action section itself. For details, refer to chapter *Assigning actions to a step* (see EcoStruxure™ Control Expert, Operating Modes).

The data structure cannot be used in variable declarations. Therefore, accessing the entire data structure (for example, as FFB parameter) is not possible.

NOTE: Usage of a SFCACTION_STATE variable in parallel branches is limited to once.

The scope of this variable to which it belongs, is the same as the SFC section where the action section has been created.

Structure of the DDT:

| Element Name | Data type | Description |
|---------------------|-----------|---|
| <i>ActionName.t</i> | TIME | Time elapsed since the action section was made active |
| <i>ActionName.q</i> | BOOL | 1: Action section active |
| | | 0: Action section inactive |

Qualifier

Introduction

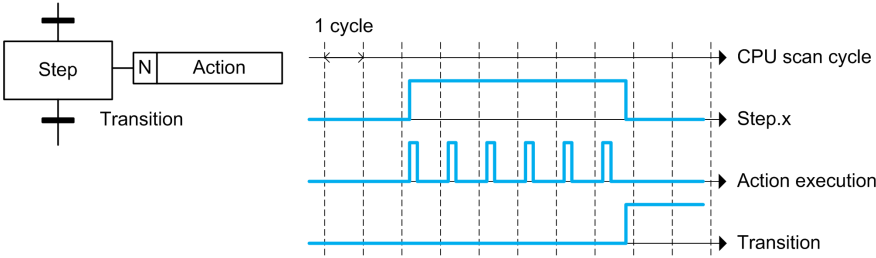
Each action that is linked to a step must have a qualifier which defines the control for that action.

NOTE: The execution behavior of an action section is dependent whether the option **SFC action behavior: Final Scan logic activated** is activated.

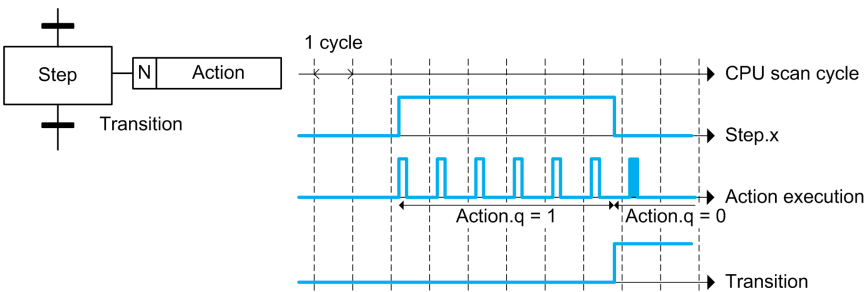
N / None Qualifiers

Meaning: None stored

Description: If the step is active, then action is 1 and if the step is inactive the action is 0.



| | |
|------------------|--|
| With final scan: | If the step is active, then action is 1 (<i>ActionName.q</i> = 1) and when the step becomes inactive the action is executed one more time with <i>ActionName.q</i> = 0. |
|------------------|--|



R Qualifier

Meaning: overriding Reset

Description: The action, which is set in another step with the qualifier S, is reset. The activation of any action can also be prevented.

NOTE: Qualifiers are automatically declared as unbuffered. This means that the value is reset to 0 after stop and cold restart, for example, when voltage is on/off. Should a buffered output be required, please use the RS or SR function block from the standard block library.

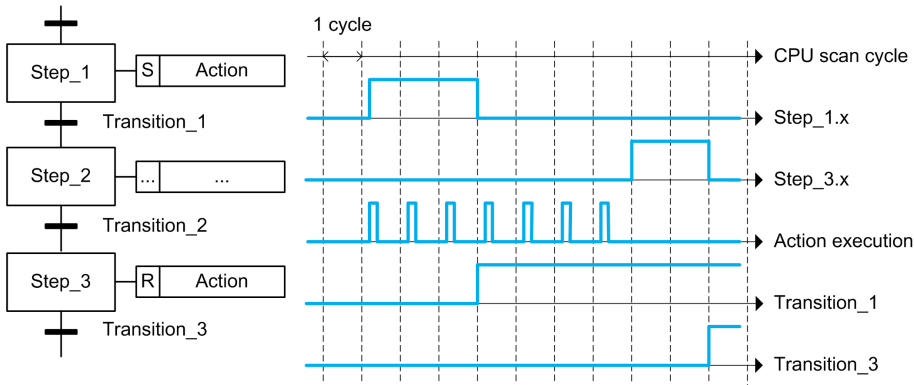
S Qualifier

Meaning: Set (saved)

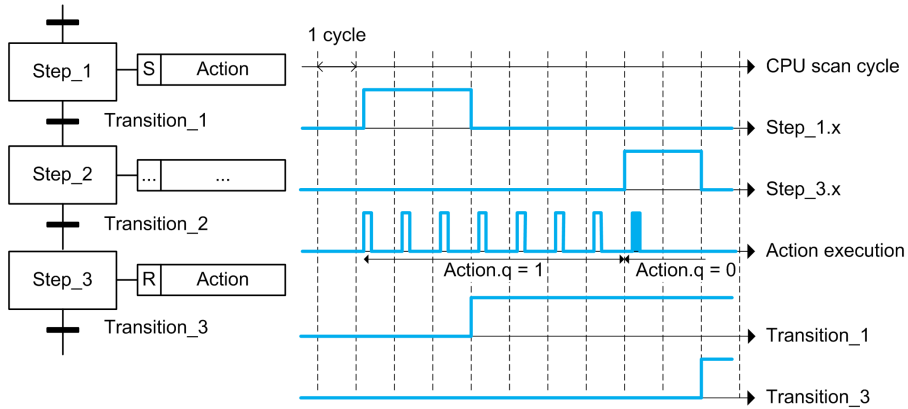
Description: The set action remains active, even when the associated step becomes inactive. The action only becomes inactive, when it is reset in another step of the current SFC section, using the qualifier R.

NOTE:

- If an action variable is modified outside of the current SFC section, it may no longer reflect the action's activation state.
- A maximum of 100 actions are permitted using the S qualifier per SFC Section.



| | |
|------------------|--|
| With final scan: | If the step is active, then action remains active ($ActionName.q = 1$) and when the action is reset in another step the action is executed one more time with $ActionName.q = 0$. |
|------------------|--|

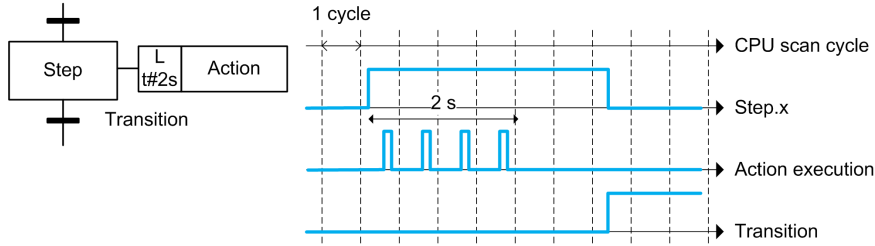


L Qualifier

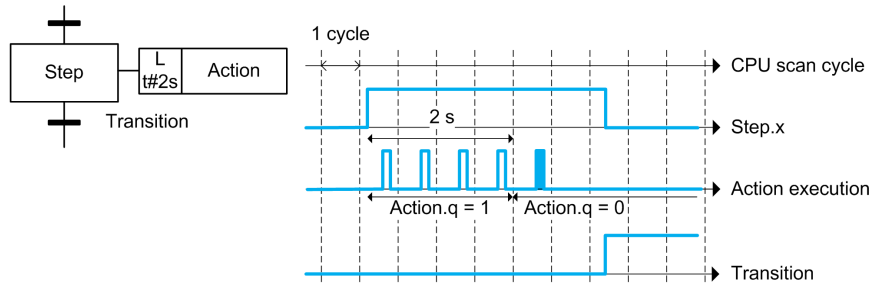
Meaning: time Limited

Description: If the step is active, the action is also active. After the process of the time duration, defined manually for the action, the action returns to 0, even if the step is still active. The action also becomes 0 if the step is inactive.

NOTE: For this qualifier, an additional duration of data type TIME must be defined.



| | |
|------------------|---|
| With final scan: | If the step is active, then action is also active (<i>ActionName</i> .q = 1). After the process of the time duration the action is executed one more time with <i>ActionName</i> .q = 0. Then the action becomes 0 even if the step is still active. |
|------------------|---|

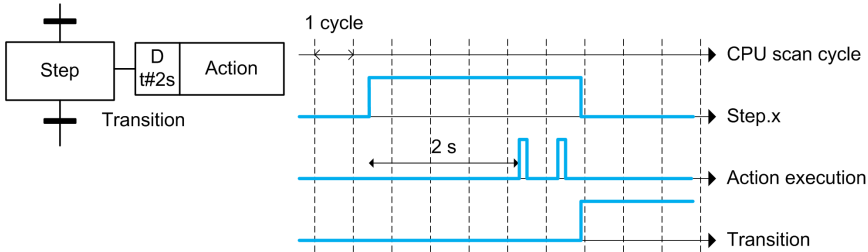


D Qualifier

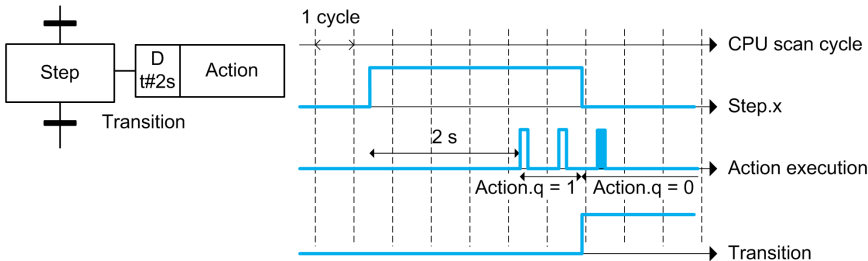
Meaning: Delayed

Description: If the step is active, the internal timer is started and the action becomes 1 after the process of the time duration, which was defined manually for the action. If the step becomes inactive after that, the action becomes inactive as well. If the step becomes inactive before the internal time has elapsed, then the action does not become active.

NOTE: For this qualifier, an additional duration of data type `TIME` must be defined.



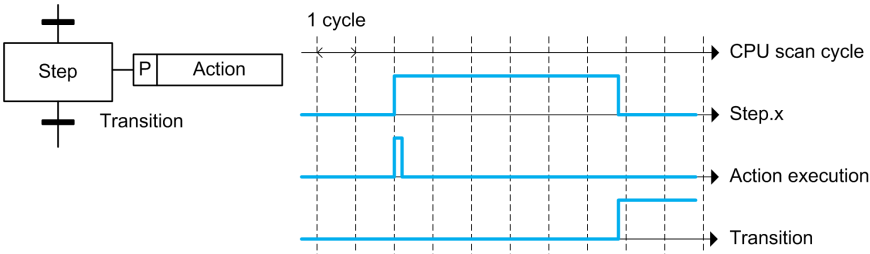
| | |
|------------------|---|
| With final scan: | If the step is active and after the process of the time duration the action becomes active (<i>ActionName.q = 1</i>) and when the step becomes inactive the action is executed one more time with <i>ActionName.q = 0</i> . |
|------------------|---|



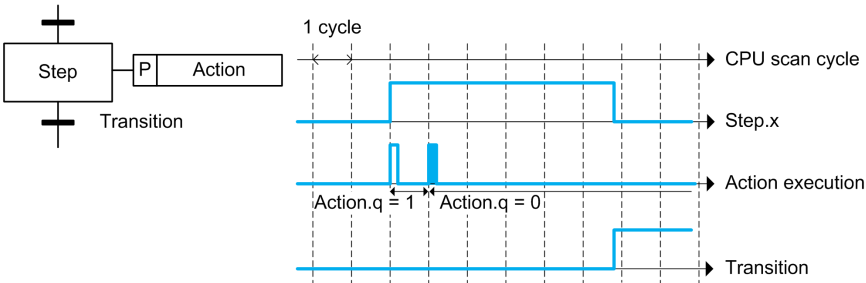
P Qualifier

Meaning: Pulse

Description: If the step becomes active, the action becomes 1 and this remains for one program cycle, independent of whether the step remains active.



| | |
|------------------|---|
| With final scan: | If the step becomes active, the action becomes 1 and this remains for two program cycles. The first one with <i>ActionName.q</i> = 1 and the second one with <i>ActionName.q</i> = 0. This is independent of whether the step remains active. |
|------------------|---|

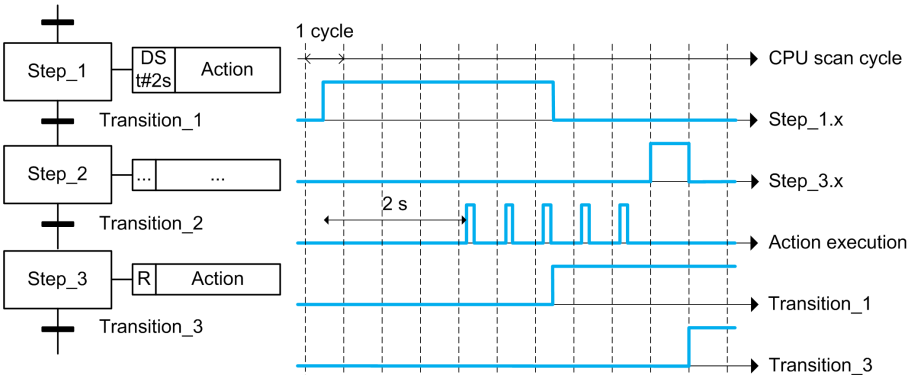


DS Qualifier

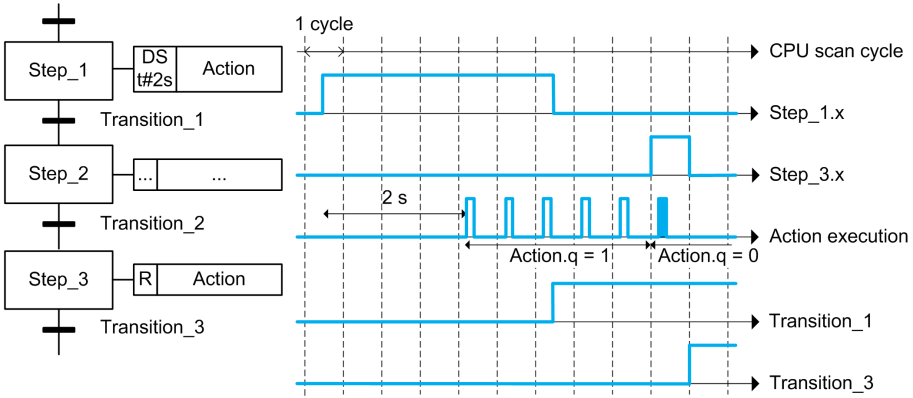
Meaning: Delayed and Set

Description: If the step becomes active, the internal timer is started and the action becomes active after the process of the manually defined time duration. The action first becomes inactive again when qualifier R is used for a reset in another step. If the step becomes inactive before the internal time has elapsed, then the action does not become active.

NOTE: For this qualifier, an additional duration of data type `TIME` must be defined.



| | |
|------------------|--|
| With final scan: | If the step is active and after the process of the time duration the action becomes active ($ActionName.q = 1$) and when the action is reset in another step, the action is executed one more time with $ActionName.q = 0$. |
|------------------|--|

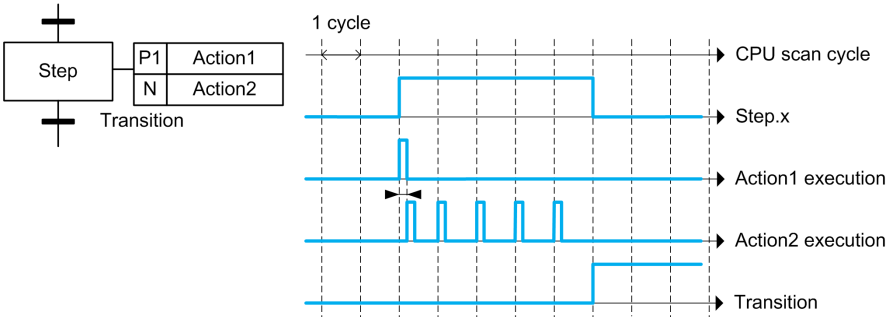


P1 Qualifier

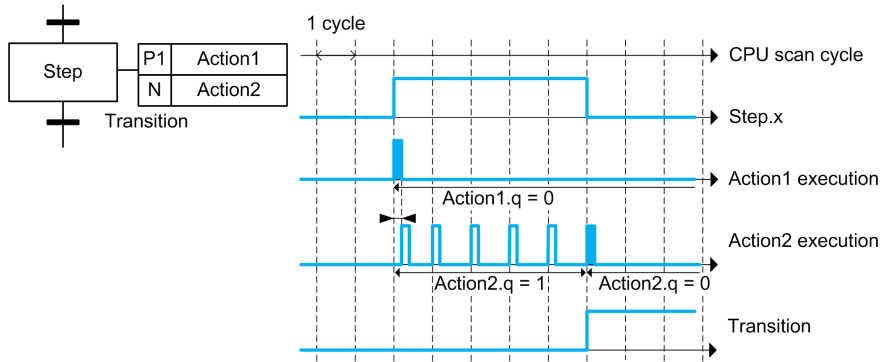
Meaning: Pulse (rising edge)

Description: If the step becomes active (0->1-edge), the action becomes 1 and this remains for one program cycle, independent of whether the step remains active.

NOTE: Independent of their position in the action list field, actions with the qualifier P1 are always processed first.



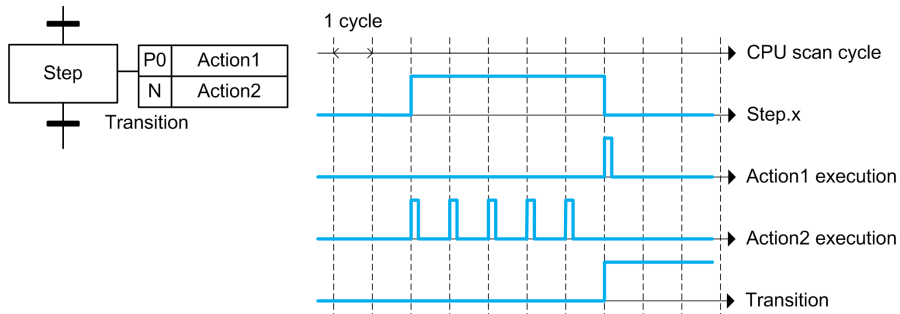
| | |
|------------------|---|
| With final scan: | If the step becomes active (0->1-edge), the action becomes 1 with $ActionName.q = 0$ and this remains for only one program cycle, independent of whether the step remains active. |
|------------------|---|



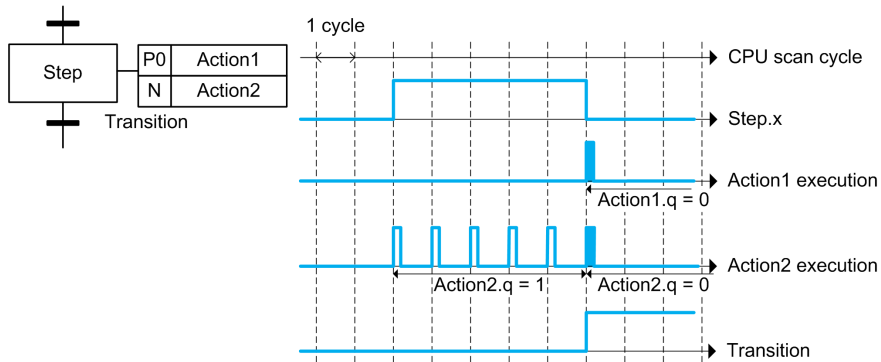
P0 Qualifier

Meaning: Pulse (falling edge)

Description: If the step becomes inactive (1->0-edge), the action becomes 1 and this remains for one program cycle.



| | |
|------------------|--|
| With final scan: | If the step becomes inactive (1->0-edge), the action becomes 1 with <i>ActionName.q = 0</i> and this remains for only one program cycle. |
|------------------|--|



Transitions and Transition Sections

Overview

This section describes the transition objects and transition sections of the SFC sequence language.

Transition

Introduction

A transition provides the condition through which the checks of one or more pre-transition steps pass on one or more consecutive steps along the corresponding link.

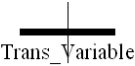

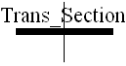
Transition Condition

Every transition is allocated with a transition condition of data type `BOOL`.

The following are authorized as transaction conditions:

- an address (input or output)
- a variable (input or output)
- a Literal or
- a Transition Section, page 343

The type of transition condition determines the position of the name.

| Transition Condition | Position of the Name |
|---|---|
| <ul style="list-style-type: none"> • Address • Variable |  |
| <ul style="list-style-type: none"> • Literal |  |
| <ul style="list-style-type: none"> • Transition Section |  |

Transition Name

If an address or a variable is used as a transition condition then the transition name is defined with that name (e.g. `%I10.4, Variable1`).

If a transition section is used as a transition condition then the section name is used as the transition name.

Transition names (maximum 32 characters) must be unique over the entire project, i.e. no other transition, variable or section (with the exception of the assigned transition section) etc., may exist with the same name. There are no case distinctions. The transition name must correspond with the standardized name conventions.

Enabling a Transition

A transition is enabled if the steps immediately preceding it are active. Transitions whose immediately preceding steps are not active are not normally analyzed.

NOTE: If no transition condition is defined, the transition will never be active.

Triggering a Transition

A transition is triggered when the transition is enabled and the associated transition conditions are satisfied.

Triggering a transition leads to the disabling (resetting) of all immediately preceding steps that are linked to the transition, followed by the activation of all immediately following steps.

Trigger Time for a Transition

The transition trigger time (switching time) can theoretically be as short as possible, but can never be zero. The transition trigger time lasts at least the duration of a program cycle.

Transition Section

Introduction

For every transition, a transition section can be created. This is a section containing the logic of the transition condition and it is automatically linked with the transition.

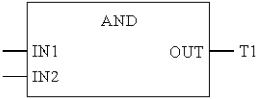

Name of Transition Section

The name of the transition section is always identical to the assigned transition, see [Transition Name](#), page 342.

Programming Languages

FBD, LD, IL and ST are possible as programming languages for transition sections.

Suggested Networks for Transition Section:

| Language | Suggested Network | Description |
|----------|---|---|
| FBD |  | <p>The suggested network contains an AND block with 2 inputs for which the output is linked with a variable having the name of the transition section.</p> <p>The suggested block can either be linked or it can be deleted if desired.</p> |
| LD |  | <p>The suggested network contains a coil which is linked with a variable having the name of the transition section.</p> <p>The suggested coil can either be linked or it can be deleted if desired.</p> |
| IL | - | <p>The suggested network is empty.</p> <p>The content may only be created of Boolean logic. The assignment of the logic result on the output (the transition variable) is done automatically, i.e. the memory assignment ST is not allowed.</p> <p>Example:</p> <pre>LD A AND B</pre> |
| ST | - | <p>The suggested network is empty.</p> <p>The content may only be created of Boolean logic in the form of a (nested) expression. The assignment of the logic result on the output (the transition variable) is done automatically, i.e. the instruction assignment := is not allowed. The expression is not terminated by a semicolon (;).</p> <p>Example:</p> <pre>A AND B or A AND (WORD_TO_BOOL (B))</pre> |

Properties of Transition Sections

Transition sections have the following properties:

- Transition sections only have one single output (transition variable), whose data type is `BOOL`. The name of these variables are identical to the names of the transition sections.

- The transition variable can only be used once in written form.
- The transition variable can be read in any position within the project.
- Only functions can be used, function blocks or procedures cannot.
- Only one coil may be used in LD.
- There is only one network, i.e. all functions used are linked with each other either directly or indirectly.
- Transition sections can only be used once.
- Transition sections belong to the SFC section in which they were defined. If the respective SFC section is deleted then all transition sections of this SFC section are also deleted automatically.
- Transition sections can be called exclusively from transitions.

Jump

Overview

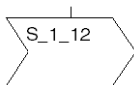
This section describes the jump objects of the SFC sequence language.

Jump

General

Jumps are used to indicate directional links that are not represented in their full length.

Representation of a jump:



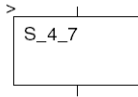
Properties of Jumps

Jumps have the following properties:

- More than one jump may have the same target step.
- In accordance with IEC 61131-3, jumps into a parallel sequence, page 348 or out of a parallel sequence are not possible.

If it should also be used again then it must be enabled explicitly.

- With jumps, there is a difference between a Sequence Jump, page 352 and a Sequence Loop, page 353.
- The jump target is indicated by the jump target symbol (>).



Jump Name

Jumps do not actually have their own names. Instead, the name of the target step (jump target) is shown inside of the jump symbol.

Link

Overview

This section describes SFC sequence language links.

Link

Introduction

Links connect steps and transitions, transitions and steps etc.

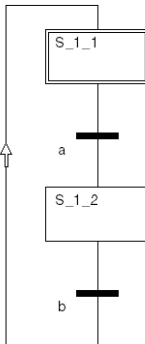
Properties of Links

Links have the following properties:

- Links between objects of the same type (step with step, transition with transition, etc.) are not possible
- Links are possible between:
 - unlinked object outputs and
 - unlinked or linked step inputs
(i.e. multiple step inputs can be linked)
- Overlapping links and other SFC objects (step, transition, jump, etc.) is not possible
- Overlapping links and links is possible
- Crossing links with links is possible and is indicated by a "broken" link:



- Links consist of vertical and horizontal segments
- Standard signal flow in a sequence string is from top to bottom. To create a loop however, links can be made from below to a step above. This applies to links from transitions, parallel branches or alternative joints to a step. In these cases, the direction of the link is indicated with an arrow symbol:



- With links, there is a difference between a String Jump, page 352 and a String Loop, page 353

Branches and Merges

Overview

This section describes the branch and merge objects of the SFC sequence language.

Alternative Branches and Alternative Joints

Introduction

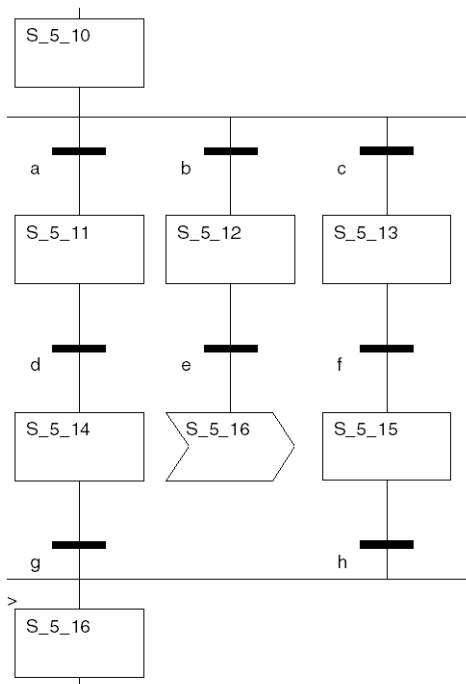
The alternative branch offers the possibility to program branches conditionally in the control flow of the SFC structure.

With alternative branches, as many transitions follow a step under the horizontal line as there are different processes.

All alternative branches are run together into a single branch again with alternative joints or Jumps, page 345 where they are processed further.

Example of an Alternative Sequence

Example of an Alternative Sequence



Properties of an Alternative Sequence

The properties of an alternative sequence mainly depend on whether the sequence control is operating in single token or multi-token mode.

See

- Properties of an Alternative Sequence in Single Token, page 351
- Properties of an Alternative Sequence in Multi Token, page 362

Parallel Branch and Parallel Joint

Introduction

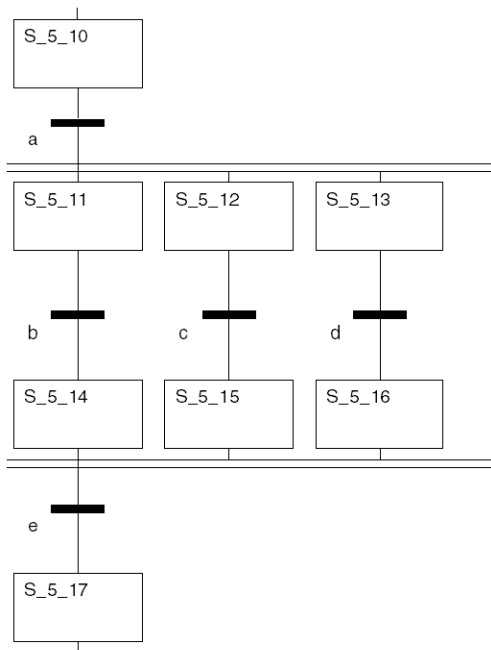
With parallel branches, switching a single transition leads to a parallel activation of more than one (maximum 32) step (branches). Execution is from left to right. After this common activation, the individual branches are processed independently from one another.

All parallel branches are grouped using a parallel joint according to IEC 61131-1. The transition following a parallel joint is evaluated when all the immediately preceding steps of the parallel joint have been set.

Combining a parallel branch with an alternative joint is only possible in Multi-Token, page 365 operation.

Example of a Parallel Sequence

Example of a Parallel Sequence



Properties of a Parallel Sequence

see

- Properties of a Parallel Sequence in Single Token, page 351
- Properties of a Parallel Sequence in Multi-Token, page 362

Text Objects

Overview

This section describes the text objects of the SFC sequence language.

Text Object

Introduction

Text can be positioned in the form of text objects using SFC sequence language. The size of these text objects depends on the length of the text. This text object is at least the size of a cell and can be vertically and horizontally enlarged to other cells according to the size of the text. Text objects can overlap with other SFC objects.

Single-Token

Overview

This section describes the "Single-Token" operating mode for sequence controls.

Execution Sequence Single-Token

Description

The following rules apply for single token:

- The original situation is defined by the initial step. The sequence string contains 1 initial step only.
- Only one step is ever active in the sequence string. The only exceptions are parallel branches in which one step is active per branch.
- The active signal status processes take place along the directional links, triggered by switching one or more transitions. The direction of the string process follows the directional links and runs from the under side of the predecessor step to the top side of the successive step.
- A transition is enabled if the steps immediately preceding it are active. Transitions whose immediately preceding steps are not active are not normally analyzed.
- A transition is triggered when the transition is enabled and the associated transition conditions are satisfied.

- Triggering a transition leads to the disabling (resetting) of all immediately preceding steps that are linked to the transition, followed by the activation of all immediately following steps.
- If more than one transition condition in a row of sequential steps has been satisfied then one step is processed per cycle.
- Steps cannot be activated or deactivated by other non-SFC sections.
- The use of macro steps is possible.
- Only one branch is ever active in alternative branches. The branch to be run is determined by the result of the transition conditions of the transitions that follow the alternative branch. If a transition condition is satisfied, the remaining transitions are no longer processed. The branch with the satisfied transition is activated. This gives rise to a left to right priority for branches. All alternative branches are combined at the end by an alternative joint or jumps.
- With parallel branches, switching a single transition leads to the activation of more than one step (branch). After this common activation, the individual branches are processed independent of one another. All parallel branches are combined at the end by a parallel joint. Jumps into a parallel branch or out of a parallel branch are not possible.

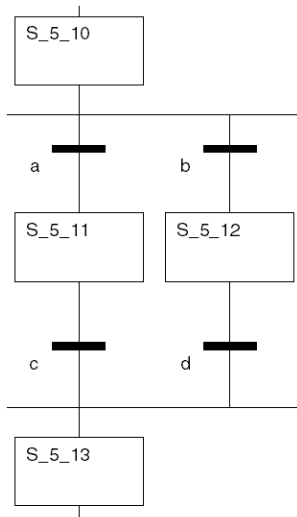
Alternative String

Alternative Strings

According to IEC 61131-3, only one switch (1-off-n-select) can be made from the transitions. The branch to be run is determined by the result of the transition conditions of the transitions that follow the alternative branch. Branch transitions are processed from left to right. If a transition condition is satisfied, the remaining transitions are no longer processed. The branch with the satisfied transition is activated. This results in a left to right priority for branches.

If none of the transitions are switched, the step that is currently set remains set.

Alternative Strings:



| If... | Then |
|--|---|
| If S_5_10 is active and transition condition a is true (independent of b), | then a sequence is run from S_5_10 to S_5_11. |
| If S_5_10 is active and transition condition b is true and a is false, | then a sequence is run from S_5_10 to S_5_12. |

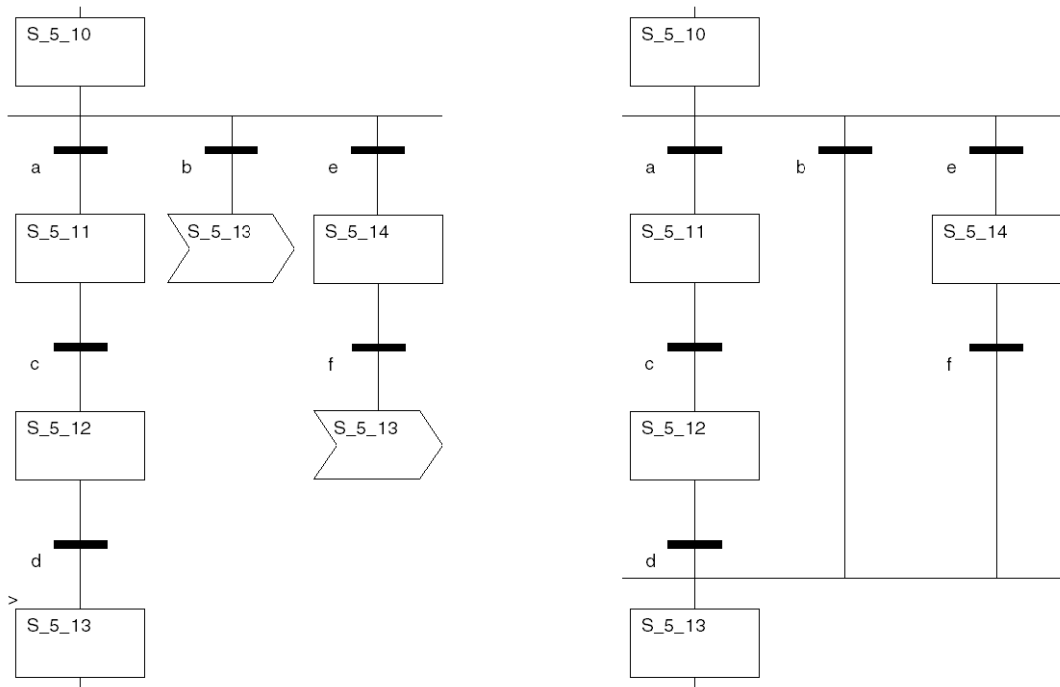
Sequence Jumps and Sequence Loops

Sequence Jump

A sequence jump is a special type of alternative branch that can be used to skip several steps of a sequence.

A sequence jump can be made with jumps or with links.

Sequence jump:



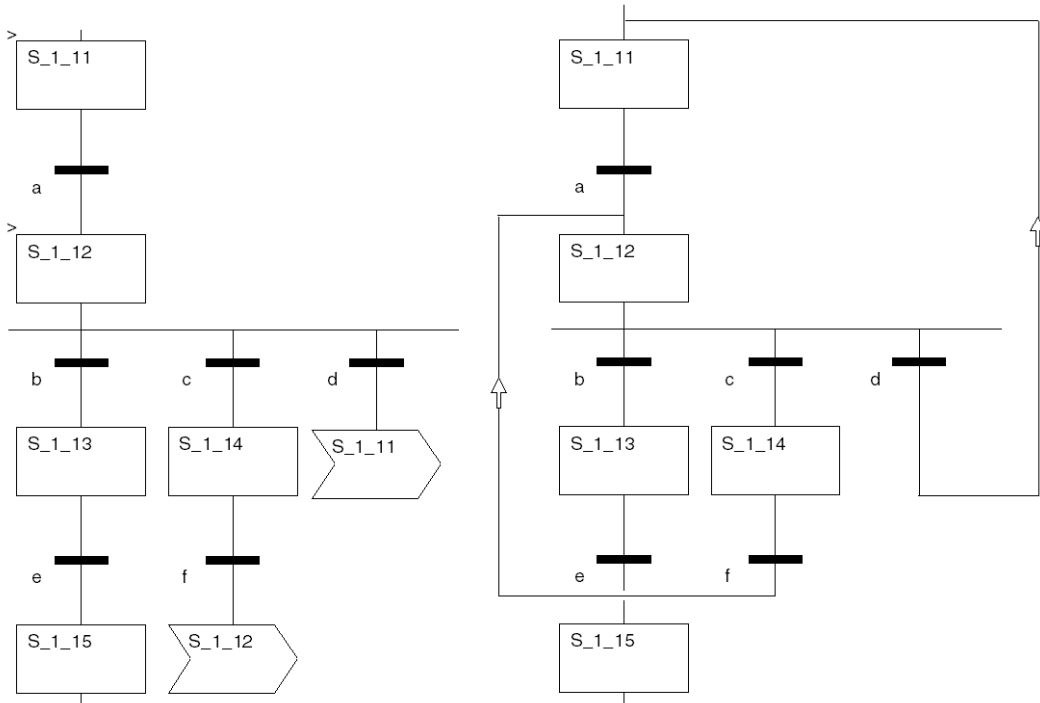
| If... | Then |
|------------------------------------|--|
| If transition condition a is true, | then a sequence is run from S_5_10 to S_5_11, S_5_12 and S_5_13. |
| If transition condition b is true, | then a jump is made from S_5_10 directly to S_5_13. |
| If transition condition e is true, | then a sequence is run from S_5_10 to S_5_14 and S_5_13. |

Sequence Loop

A sequence loop is a special type of alternative branch with which one or more branches lead back to a previous step.

A sequence loop can be made with jumps or with links.

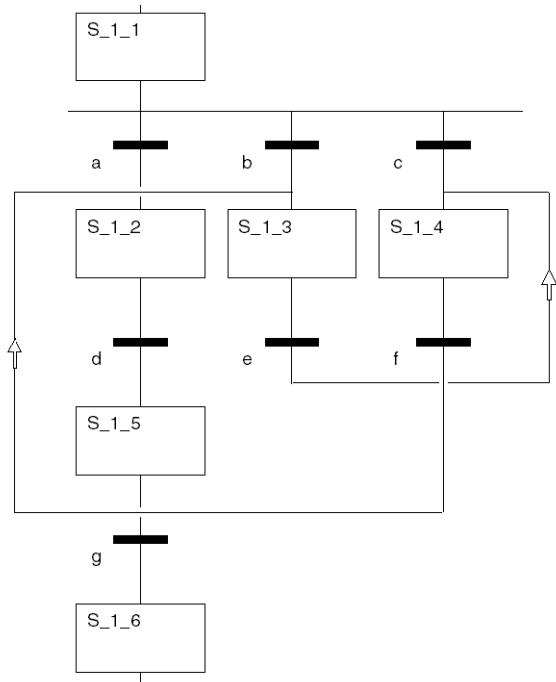
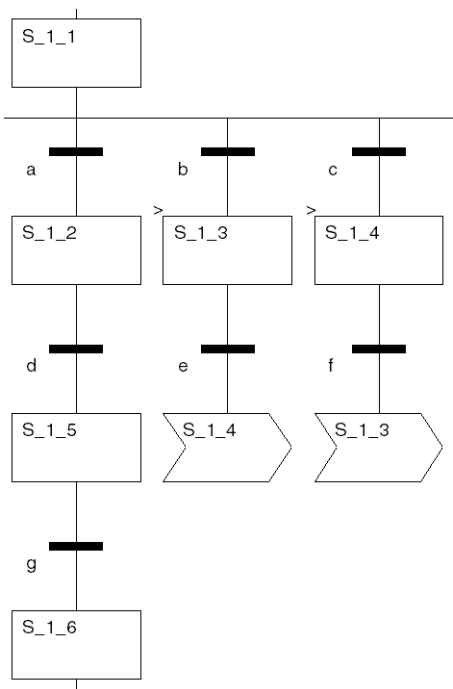
Sequence loop:



| If... | Then |
|---|--|
| If transition condition a is true, | then a sequence runs from S_1_11 to S_1_12. |
| If transition condition b is true, | then a sequence runs from S_1_12 to S_1_13. |
| If transition condition b is false and c is true, | then a sequence runs from S_1_12 to S_1_14. |
| If transition condition f is true, | then a jump is made from S_1_14 back to S_1_12. |
| The loop from S_1_12 by means of transition conditions c and f back to S_1_12 is repeated until transition condition b is true or c is false and d is true. | |
| If transition conditions b and c are false and d is true, | then a jump is made from S_1_12 directly back to S_1_11. |
| The loop from S_1_11 to S_1_12 and back to S_1_11 via transition conditions a and d is repeated until transition condition b or c is true. | |

Infinite sequence loops are not permitted within an alternative sequence.

Infinite sequence loops:



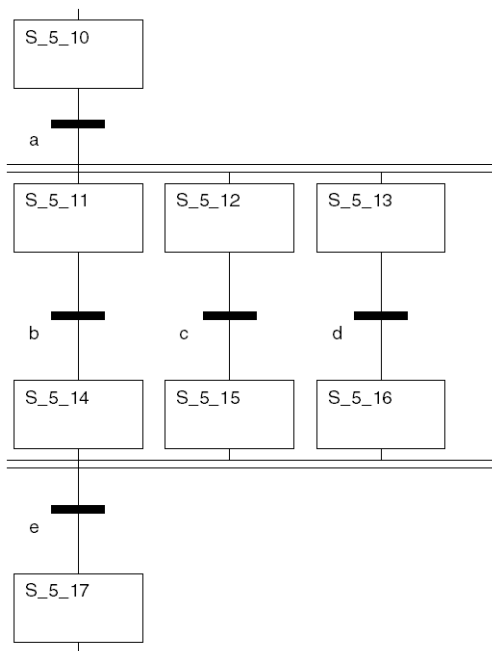
| If... | Then |
|---|---|
| If transition condition b is true, | then a sequence runs from S_1_1 to S_1_3. |
| If transition condition e is true, | then a jump is made to S_1_4. |
| If transition condition f is true, | then a jump is made to S_1_3. |
| The loop from S_1_3 via transition condition e, to S_1_4 via transition condition f and a jump back to S_1_3 again, is now repeated infinitely. | |

Parallel Strings

Parallel Strings

With parallel branches, switching a single transition leads to a parallel activation of more than one (maximum 32) steps (branches). This applies with Single-Token as well as with Multi-Token.

Processing Parallel Strings:

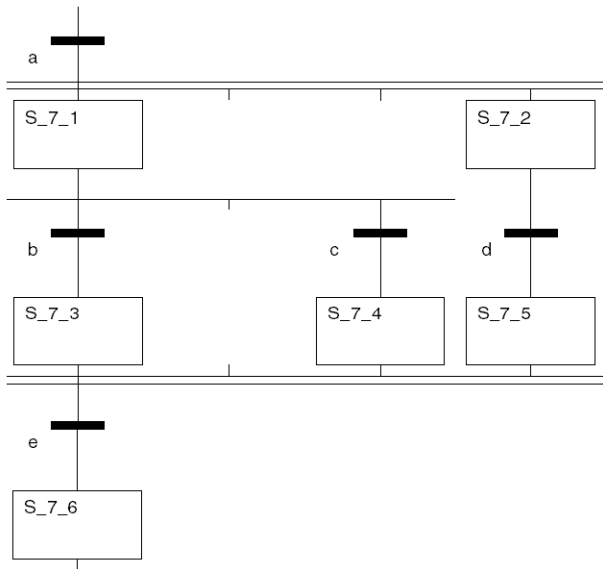


| If... | Then |
|---|--|
| If S_5_10 is active and transition condition a, which belongs to the common transition, is also true, | then a sequence runs from S_5_10 to S_5_11, S_5_12 and S_5_13. |
| If steps S_5_11, S_5_12 and S_5_13 are activated, | then the strings run independently of one another. |
| If S_5_14, S_5_15 and S_5_16 are active at the same time and transition condition e, which belongs to the common transition, is true, | then a sequence is run from S_5_14, S_5_15 and S_5_16 to S_5_17. |

Using an Alternative Branch in a Parallel String

If a single alternative branch is used in a parallel string, it leads to blocking the string with Single-Token.

Using an Alternative Branch in a Parallel String:



| If... | Then |
|---|--|
| If transition condition a is true, | then a sequence is run to s_7_1 and s_7_2. |
| If steps s_7_1 and s_7_2 are activated, | then the strings run independently of one another. |
| If transition condition d is true, | then a sequence runs to s_7_5. |
| If transition condition b is true and c is false, | then a sequence runs to s_7_3. |
| <p>Since s_7_3, s_7_4 and s_7_5 are linked with a parallel merge, no sequence can follow to s_7_6 because s_7_3 and s_7_4 can never be active at the same time.</p> <p>(Either s_7_3 is activated with transition condition b or s_7_4 with transition condition c, never both at the same time.)</p> <p>Therefore s_7_3, s_7_4 and s_7_5 can never be active at the same time either. The string is blocked.</p> <p>The same problem occurs if transition condition b is false and c is true when entering the alternative branch.</p> | |

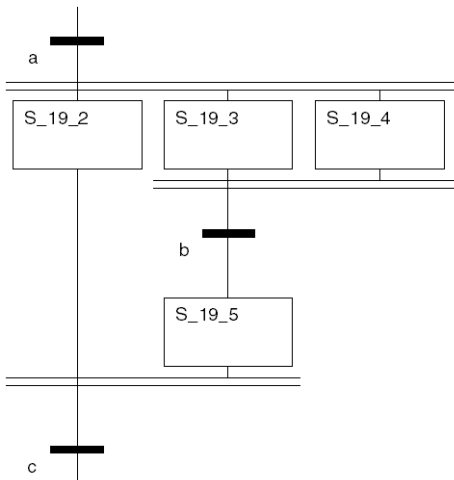
Asymmetric Parallel String Selection

Introduction

According to IEC 61131-3, a parallel branch must always be terminated with a parallel merge. The number of parallel branches must not coincide with the number of parallel merges however.

Greater Amount of Merges

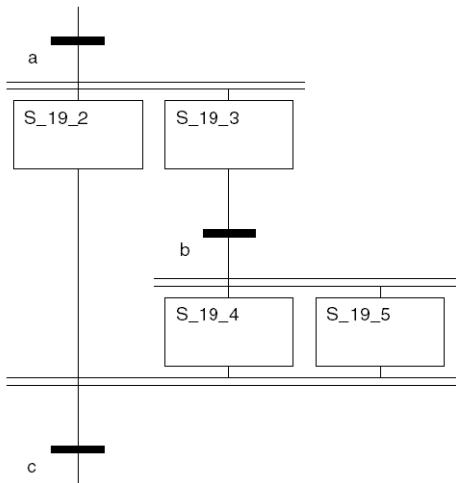
String with 1 Parallel Branch and 2 Parallel Merges:



| If... | Then |
|--|--|
| If transition condition a is true, | then a sequence runs to S_19_2, S_19_3 and S_19_4. |
| If steps S_19_2, S_19_3 and S_19_4 are activated, | then the strings run independently of one another. |
| If transition condition b is true, | then a sequence runs to S_19_5. |
| If steps S_19_2 and S_19_5 are active and transition condition c, is true, | then the parallel string is departed. |

Greater Amount of Branches

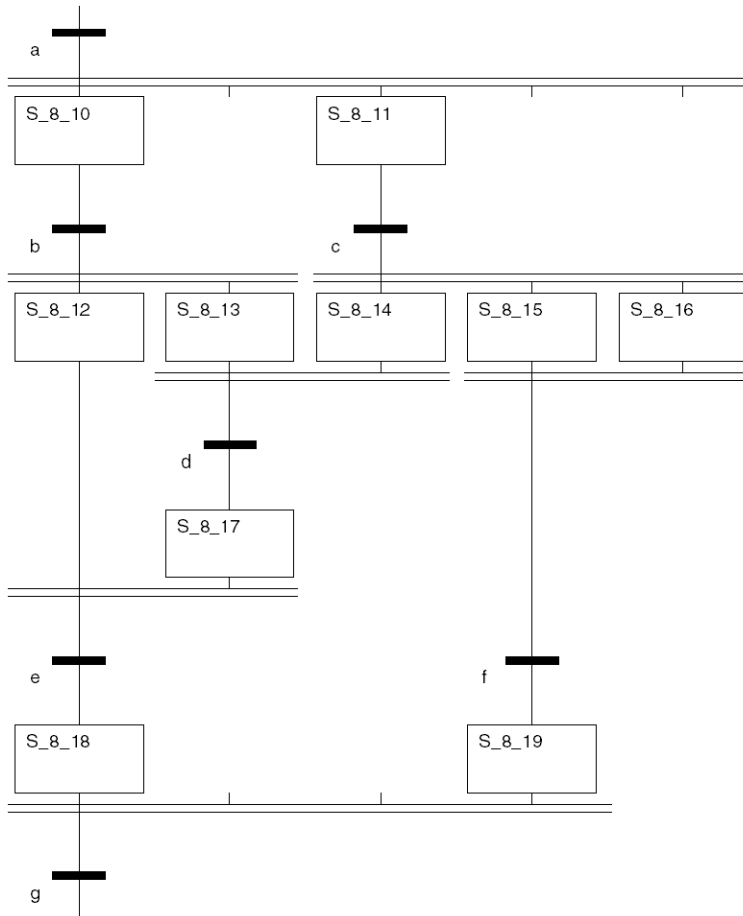
String with 2 Parallel Branches and 1 Parallel Merge:



| If... | Then |
|--|--|
| If transition condition a is true, | then a sequence runs to S_19_2 and S_19_3. |
| If steps S_19_2 and S_19_3 are activated, | then the strings run independently of one another. |
| If transition condition b is true, | then a sequence runs to S_19_4 and S_19_5. |
| If steps S_19_4 and S_19_5 are activated, | then the strings run independently of one another. |
| If steps S_19_2, S_19_4 and S_19_5 are active and transition condition c is true, | then the parallel string is departed. |

Nested Parallel Strings

Nested Parallel Strings:



| If... | Then |
|--|--|
| If transition condition <i>a</i> is true, | then a sequence runs to S_8_10 and S_8_11. |
| If transition condition <i>b</i> is true, | then a sequence runs to S_8_12 and S_8_13. |
| If transition condition <i>c</i> is true, | then a sequence runs to S_8_14, S_8_15 and S_8_16. |
| If steps S_8_13 and S_8_14 are active and transition condition <i>d</i> , is true, | then a sequence runs to S_8_17. |

| If... | Then |
|--|---------------------------------|
| If steps S_8_12 and S_8_17 are active and transition condition e, is true, | then a sequence runs to S_8_18. |
| ... | ... |

Multi-Token

Overview

This section describes the "Multi-Token" operating mode for sequence controls.

Multi-Token Execution Sequence

Description

The following rules apply for Multi-Token:

- The original situation is defined in a number of initial steps (0 to 100) which can be defined.
- A number of steps which can be freely defined can be active at the same time in a sequence string.
- The active signal status processes take place along the directional links, triggered by switching one or more transitions. The direction of the string process follows the directional links and runs from the under side of the predecessor step to the top side of the successive step.
- A transition is enabled if the steps immediately preceding it are active. Transitions whose immediately preceding steps are not active are not analyzed.
- A transition is triggered when the transition is enabled and the associated transition conditions are satisfied.
- Triggering a transition leads to the disabling (resetting) of all immediately preceding steps that are linked to the transition, followed by the activation of all immediately following steps.
- If more than one transition condition in a row of sequential steps has been satisfied then one step is processed per cycle.
- Steps and macro steps can be activated or deactivated by other non-SFC sections or by user operations.
- If an active step is activated and deactivated at the same time then the step remains active.

- The use of macro steps is possible. Whereas the macro step section can also contain initial steps.
- More than one branch can be active with alternative branches. The branches to be run are determined by the result of the transition conditions of the transitions that follow the alternative branch. Branch transitions are processed in parallel. The branches with satisfied transitions are activated. All alternative branches do not have to be combined at the end by an alternative joint or jumps.
- If jumps are to be made into a parallel branch or out of a parallel branch then this option can be enabled. All parallel branches do not have to be combined at the end by a parallel joint in this case.
- Subroutine calls be used in an action section.
- Multiple tokens can be created with:
 - Multiple initial steps
 - Alternative or parallel branches that are not terminated
 - Jumps in combination with alternative and parallel strings
 - Activation of steps using the SFC control block `SETSTEP` from a non -SFC section or with SFC control instructions
- Tokens can be ended with:
 - Simultaneous meeting of two or more tokens in a step
 - Deactivation of steps using the SFC control block `RESETSTEP` from a non -SFC section or with SFC control instructions

Alternative String

Alternative Strings

The user can define the behavior for the evaluation of transition conditions in alternative branches with Multi-Token.

The following are possible:

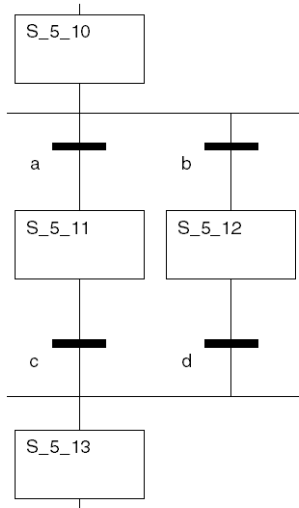
- Processing is from left to right with a stop after the first active transition (1-off-n-select). This corresponds with the behavior of alternative strings with Single-Token, page 351.
- Parallel processing of all transitions of the alternative branch (x-off-n-select)

x-off-n-select

With Multi-Token, more than one parallel switch can be made from the transitions (1-off-n-select). The branches to be run are determined by the result of the transition conditions of the transitions that follow the alternative branch. The transitions of the branches are all processed. All branches with satisfied transitions are activated.

If none of the transitions are switched, the step that is currently set remains set.

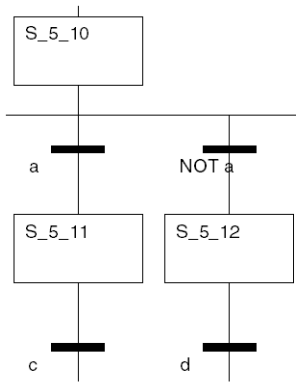
x-off-n-select:



| If... | | Then | |
|--|---|--|---|
| If S_5_10 is active and transition condition a is true and b is false, | | then a sequence is run from S_5_10 to S_5_11. | |
| If S_5_10 is active and transition condition a is false and b is true, | | then a sequence is run from S_5_10 to S_5_12. | |
| If S_5_10 is active and transition conditions a and b are true, | | then a sequence is run from S_5_10 to S_5_11 and S_5_12. | |
| A second token is created by the parallel activation of the two alternative branches. These two tokens are now running parallel to one another, i.e. S_5_11 and S_5_12 are active at the same time. | | | |
| Token 1 (S_5_11) | | Token 2 (S_5_12) | |
| If... | Then | If... | Then |
| If the transition condition c is true, | then a sequence is run from S_5_11 to S_5_13. | If transition condition d is true, | then a sequence is run from S_5_12 to S_5_13. |
| If S_5_13 is still active (token 1) because of the activation of transition condition c, then token 2 is ended and the string will be further processed as Single-Token. If S_5_13 is no longer active (token 1), then it is reactivated by token 2 and both tokens continue running parallel (Multi-Token). | | | |

If alternative branches should only be switched exclusively in this mode of operation, then this must be defined explicitly with the transition logic.

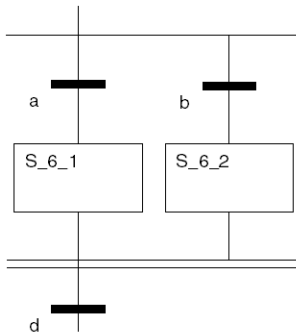
Example:



Terminating an Alternative Branch with a Parallel Merge

If a parallel merge is used to terminate an alternative branch, it may block the string.

Terminating an Alternative Branch with a Parallel Merge:



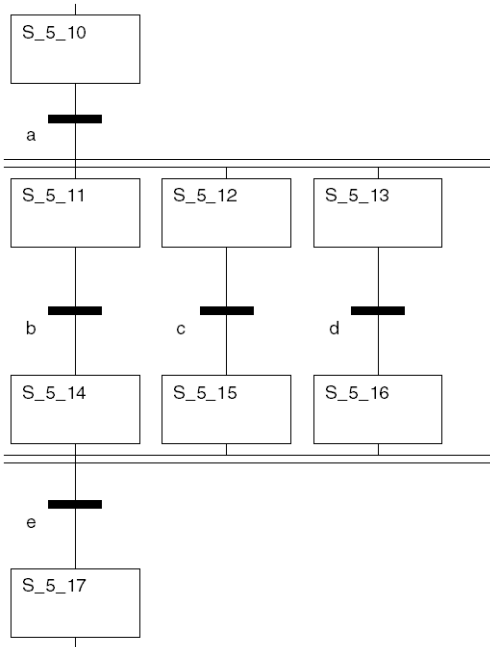
| If... | Then |
|---|--------------------------------|
| If transition condition a is true and b is false, | then a sequence runs to S_6_1. |
| <p>Since S_6_1 and S_6_2 are linked by a parallel merge, the branch cannot be departed because S_6_1 and S_6_2 can never be active at the same time.</p> <p>(Either S_6_1 is activated with transition condition a or S_6_2 with transition condition b.)</p> <p>Therefore S_6_1 and S_6_2 can never be active at the same time either. The string is blocked.</p> <p>This block can be removed, for example, by a second timed token that runs via transition b.</p> | |

Parallel Strings

Parallel Strings

With parallel branches, switching a single transition leads to a parallel activation of more than one (maximum 32) steps (branches). This applies with Single-Token as well as with Multi-Token

Processing Parallel Strings:

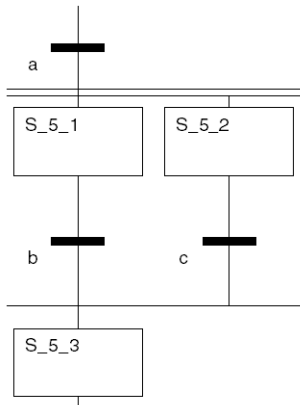


| If... | Then |
|---|--|
| If S_5_10 is active and transition condition a, which belongs to the common transition, is also true, | then a sequence runs from S_5_10 to S_5_11, S_5_12 and S_5_13. |
| If steps S_5_11, S_5_12 and S_5_13 are activated, | then the strings run independently of one another. |
| If S_5_14, S_5_15 and S_5_16 are active at the same time and transition condition e, which belongs to the common transition, is true, | then a sequence is run from S_5_14, S_5_15 and S_5_16 to S_5_17. |

Terminating a Parallel Branch with an Alternative Merge

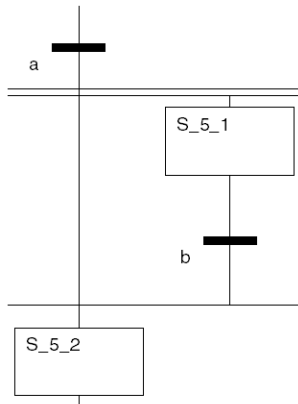
Terminating a parallel branch can also be done with an alternative merge instead of a parallel merge with Multi-Token.

Terminating a Parallel String with an Alternative Branch (variation 1):



| | | | |
|--|--|--|--------------------------------|
| If... | Then | | |
| If the transition condition a is true, | then a sequence runs to s_5_1 and s_5_2. | | |
| If steps s_5_1 and s_5_2 are activated, | then the strings run independently of one another. | | |
| If transition condition b is true and c is false, | then a sequence runs to s_5_3. | | |
| A second token is created by the sequence running on the alternative merge out of the parallel string. The two tokens are running parallel to one another, i.e. s_5_2 and s_5_3 are active at the same time. | | | |
| Token 1 (S_5_3) | | Token 2 (S_5_2) | |
| If... | Then | If... | Then |
| Step s_5_3 is active. | | Step s_5_2 is active. | |
| | | If the transition condition c is true, | then a sequence runs to s_5_3. |
| If s_5_3 is still active (token 1) then token 2 is ended and the string is further processed as Single-Token. | | | |
| If s_5_3 is no longer active (token 1), then it is reactivated by token 2 and both tokens continue running parallel (Multi-Token). | | | |

Terminating a Parallel String with an Alternative Branch (variation 2):

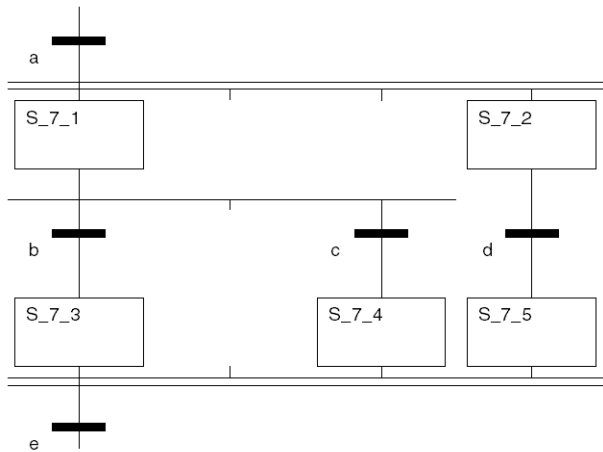


| | | | |
|--|-------------|--|--------------------------------|
| If... | | Then | |
| If the transition condition a is true, | | then a sequence runs to S_5_1 and S_5_2. | |
| A second token is created by the sequence running on the alternative merge out of the parallel string. These two tokens are now running parallel to one another, i.e. S_5_1 and S_5_2 are active at the same time. | | | |
| Token 1 (S_5_2) | | Token 2 (S_5_1) | |
| If... | Then | If... | Then |
| Step S_5_2 is active. | | Step S_5_1 is active. | |
| | | If transition condition b is true, | then a sequence runs to S_5_2. |
| If S_5_2 is still active (token 1) then token 2 is ended and the string is further processed as Single-Token. | | | |
| If S_5_2 is no longer active (token 1), then it is reactivated by token 2 and both tokens continue running parallel (Multi-Token). | | | |

Using an Alternative Branch in a Parallel String

If one single alternative branch is used in a parallel string, it may block the string.

Using an Alternative Branch in a Parallel String:



| If... | Then |
|---|--|
| If transition condition a is true, | then a sequence is run to s_7_1 and s_7_2. |
| If steps s_7_1 and s_7_2 are activated, | then the strings run independently of one another. |
| If transition condition d is true, | then a sequence runs to s_7_5. |
| If transition condition b is true, | then a sequence runs to s_7_3. |
| Since s_7_3, s_7_4 and s_7_5 are linked by a parallel merge, the parallel string cannot be departed because s_7_3 and s_7_4 can never be active at the same time. (Either s_7_3 is activated with transition condition b or s_7_4 with transition condition c.) Therefore s_7_3, s_7_4 and s_7_5 cannot be active at the same time either. The string is blocked. This block can be removed for example, by a second timed token that runs via transition c. | |

Jump into a Parallel String

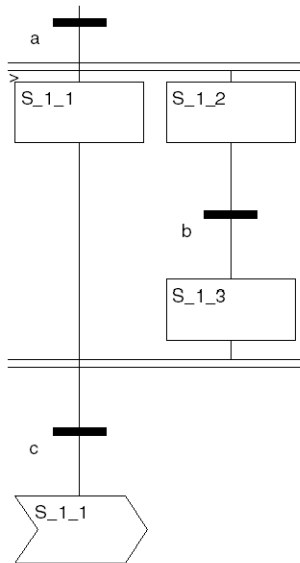
Description

The ability to jump into a parallel string or out of a parallel string can be enabled optionally with multi-token

A jump into a parallel string does not activate all branches. Since the transition after the parallel joint is only evaluated if all steps which directly precede the transition are set, the parallel string can no longer be departed, the string is blocking.

Jump into a Parallel String

Jump into a Parallel String



| If... | Then |
|---|---|
| If the transition condition a is true, | then a sequence runs to S_1_1 and S_1_2. |
| If steps S_1_1 and S_1_2 are activated, | then the strings run independently of one another. |
| If S_1_2 is active and transition condition b, is true, | then a sequence runs from S_1_2 to S_1_3. |
| If S_1_1 and S_1_3 are active and transition condition c, which belongs to the common transition, is true, | then a sequence runs from S_1_1 and S_1_3 to a jump to S_1_1. |
| If S_1_1 is activated by the jump, | then only the branch from S_1_1 is active. The branch from S_1_2 is not active. |
| Since S_1_1 and S_1_3 are not active at the same time, the string cannot continue. The string is blocked. This block can be removed by e.g. a second timed token that is set to reactivate step S_1_2. | |

Jump out of a Parallel String

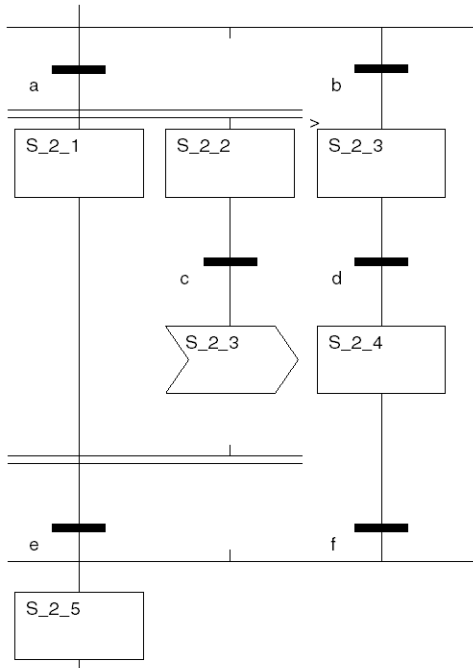
Introduction

The ability to jump into a parallel string or out of a parallel string can be enabled optionally with multi-token

Extra tokens are generated in all cases.

Jump out of a Parallel String

Jump out of a Parallel String:

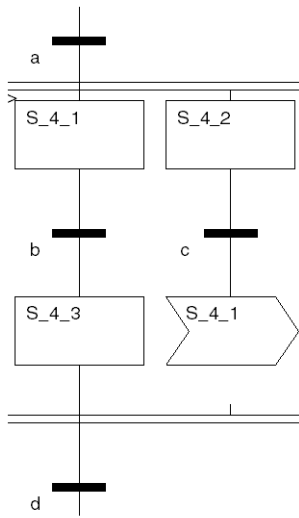


| | | | |
|--|--------------------------------|--|--------------------------------|
| If... | | Then | |
| If the transition condition a is true and b is false, | | then a sequence runs to s_2_1 and s_2_2. | |
| If steps s_2_1 and s_2_2 are activated, | | then the strings run independently of one another. | |
| If the transition condition c is true, | | then a jump is made to s_2_3. | |
| A second token is created by the jump out of the parallel string. Both tokens are running parallel to one another, i.e. s_2_1 and s_2_3 are active at the same time. | | | |
| Token 1 (S_2_1) | | Token 2 (S_2_3) | |
| If... | Then | If... | Then |
| If the transition condition e is true, | then a sequence runs to s_2_5. | If transition condition d is true, | then a sequence runs to s_2_4. |

| | | |
|---|--------------------------------------|--------------------------------------|
| | If transition condition f is true, | then a sequence runs to S_{2_5} . |
| <p>If S_{2_5} is still active (token 1) because of the activation of transition condition e, then token 2 is ended and the string will be further processed as Single-Token.</p> <p>If S_{2_5} is no longer active (token 1), then it is reactivated by token 2 and both tokens continue running parallel (Multi-Token).</p> | | |

Jump Between Two Branches of a Parallel String

Jump Between Two Branches of a Parallel String:

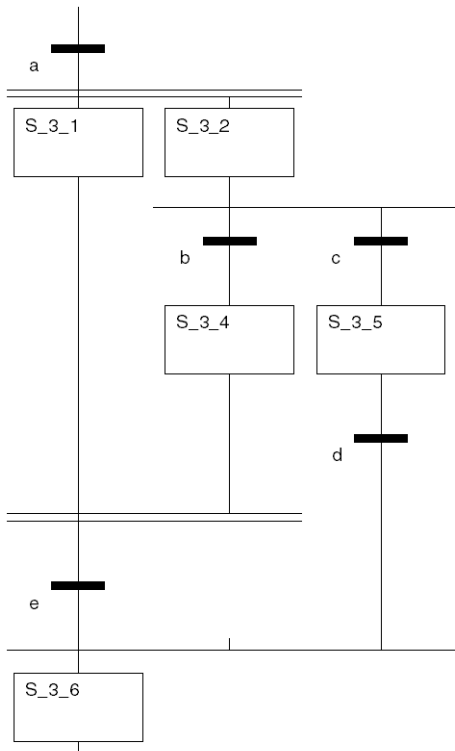


| | | | |
|--|---|------------------------------|-------------|
| If... | Then | | |
| If the transition condition a is true, | then a sequence runs to S_{4_1} and S_{4_2} . | | |
| If steps S_{4_1} and S_{4_2} are activated, | then the strings run independently of one another. | | |
| If transition condition b is true, | then a sequence runs to S_{4_3} . | | |
| If the transition condition c is true, | then a jump is made to S_{4_1} . | | |
| A second token is created by the jump out of a branch string. Both tokens are running parallel to one another, i.e. S_{4_3} and S_{4_1} are active at the same time. | | | |
| Token 1 (S_{4_3}) | Token 2 (S_{4_1}) | | |
| If... | Then | If... | Then |
| Step S_{4_3} is processed | | Step S_{4_1} is processed | |

| | | |
|---|---|--|
| | If transition condition <i>b</i> is true, | then a sequence runs to <i>S_4_3</i> . |
| <p>If step <i>S_4_3</i> is still active (token 1) during the activation by token 2 then token 2 is ended and the string will continue to be processed as Single-Token.</p> <p>If step <i>S_4_3</i> is no longer active (token 1) because of the activation by token 2, then it is reactivated by token 2 and both tokens continue running parallel (Multi-Token).</p> <p>In both cases, true transition condition <i>d</i> causes the parallel string to be left.</p> | | |

Leaving a Parallel String with an Alternative Branch

Leaving a Parallel String with an Alternative Branch:



| If... | Then |
|---|---|
| If the transition condition <i>a</i> is true, | then a sequence runs to <i>S_3_1</i> and <i>S_3_2</i> . |
| If steps <i>S_3_1</i> and <i>S_3_2</i> are activated, | then the strings run independently of one another. |
| If transition condition <i>b</i> is false and <i>c</i> is true, | then a sequence runs to <i>S_3_5</i> . |

| | | | |
|--|-------------|---|--------------------------------------|
| A second token is created by the sequence running on the alternative branch out of the parallel string. Both tokens are running parallel to one another, i.e. s_{3_1} and s_{3_5} are active at the same time. | | | |
| Token 1 (s_{3_1}) | | Token 2 (s_{3_5}) | |
| If... | Then | If... | Then |
| Since s_{3_4} cannot become active, s_{3_1} remains (token 1) active. | | If transition condition d is true, | then a sequence runs to s_{3_6} . |
| If transition condition a is true then a sequence runs to s_{3_1} and s_{3_2} . This ends token 2 and the string is again processed as Single-Token. | | | |
| If the transition condition a is true, then a sequence runs to s_{3_1} and s_{3_2} . | | | |
| | | If transition condition b is true and c is false, | then a sequence runs to s_{3_4} . |
| Since s_{3_4} cannot become active, s_{3_1} remains (token 1) active until a sequence appears on s_{3_2} (token 2) and the transition is b . | | | |
| If s_{3_4} is no longer active (token 1), then it is reactivated by token 2 and both tokens continue running parallel (Multi-Token). | | | |
| (Merging the two tokens can also be done in s_{4_3} .) | | | |

Instruction List (IL)

What's in This Chapter

| | |
|---|-----|
| General Information about the IL Instruction List..... | 374 |
| Calling Elementary Functions, Elementary Function Blocks, Derived Function Blocks and Procedures | 395 |

Overview

This chapter describes the programming language instruction list IL which conforms to IEC 61131.

General Information about the IL Instruction List

Overview

This section contains a general overview of the IL instruction list.

General Information about the IL Instruction List

Introduction

Using the Instruction list programming language (IL), you can call function blocks and functions conditionally or unconditionally, perform assignments and make jumps conditionally or unconditionally within a section.

Instructions

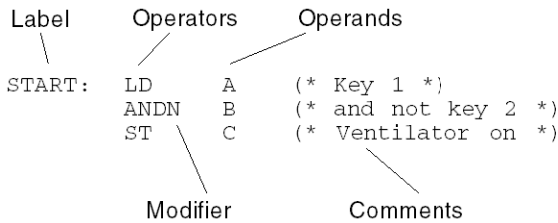
An instruction list is composed of a series of instructions.

Each instruction begins on a new line and consists of:

- an Operator, page 381,
- if necessary with a Modifier, page 379 and
- if necessary one or more Operands, page 377

Should several operands be used, they are separated by commas. It is possible for a Label, page 393 to be in front of the instruction. This label is followed by a colon. A Comment, page 395 can follow the instruction.

Example:



Structure of the Programming Language

IL is a so-called accumulator orientated language, i.e. each instruction uses or alters the current content of the accumulator (a form of internal cache). IEC 61131 refers to this accumulator as the "result".

For this reason, an instruction list should always begin with the LD operand ("Load in accumulator command").

Example of an addition:

| Command | Meaning |
|---------|--|
| LD 10 | Load the value 10 into the accumulator. |
| ADD 25 | "25" is added to the contents of the accumulator. |
| ST A | The result is stored in the variable A. The content of the variable A and the accumulator is now 35. Any further instruction will work with accumulator contents "35" if it does not begin with LD. |

Compare operations likewise always refer to the accumulator. The Boolean result of the comparison is stored in the accumulator and therefore becomes the current accumulator content.

Example of a comparison:

| Command | Meaning |
|---------|--|
| LD B | The value B is loaded into the accumulator. |
| GT 10 | 10 is compared with the contents of the accumulator. |
| ST A | The result of the comparison is stored in the variable A. If B is less than or equal to 10, the value of both variable A and the accumulator content is 0 (FALSE). If B is greater than 10, the value of both variable A and the accumulator content is 1 (TRUE). |

Section Size

The length of an instruction line is limited to 300 characters.

The length of an IL section is not limited within the programming environment. The length of an IL section is usually limited by the size of the PLC memory.

NOTE: There is no size limitation for section, but sometime when using a large amount of literal assignments or some specific instructions, a section can generate a Code generation failure during an application build. Then the solution is to split the section in two or more sections to build the application.

Syntax

Identifiers and Keywords are not case sensitive.

Spaces and tabs have no influence on the syntax and can be used as and when required,

Exception: Not allowed - spaces and tabs

- keywords
- literals
- values
- identifiers
- variables and
- limiter combinations [e.g. (* for comments)]

Execution Sequence

Instructions are executed line by line, from top to bottom. This sequence can be altered with the use of parentheses.

If, for example, A, B, C and D have the values 1, 2, 3 and 4, and are calculated as follows:

```
LD A
ADD B
SUB C
MUL C
ST E
```

the result in E will be 0.

In the case of the following calculation:

```
LD A
ADD B
SUB (
LD C
```



```
MUL D  
)  
ST E
```

the result in `E` will be -9.

Error Behavior

The following conditions are handled as an error when executing an expression:

- Attempting to divide by 0.
- Operands do not contain the correct data type for the operation.
- The result of a numerical operation exceeds the value range of its data type

IEC Conformity

For a description of IEC conformity for the IL programming language, see IEC Conformity, page 508.

Operands

Introduction

Operators are used for operands.

An operand can be:

- an address
- a literal
- a variable
- a multi-element variable
- an element of a multi-element variable
- an EFB/DFB output or
- an EFB/DFB call

Data Types

The operand and the current accumulator content must be of the same type. Should operands of various types be processed, a type conversion must be performed beforehand.

In the example the integer variable `i1` is converted into a real variable before being added to the real variable `r4`.

```
LD i1
INT_TO_REAL
ADD r4
ST r3
```

As an exception to this rule, variables with data type `TIME` can be multiplied or divided by variables with data type `INT`, `DINT`, `UINT` or `UDINT`.

Permitted operations:

- LD timeVar1
DIV dintVar1
ST timeVar2
- LD timeVar1
MUL intVar1
ST timeVar2
- LD timeVar1
MUL 10
ST timeVar2

This function is listed by IEC 61131-3 as "undesired" service.

Direct Use of Addresses

Addresses can be used directly (without a previous declaration). In this case the data type is assigned to the address directly. The assignment is made using the "Large prefix".

The different large prefixes are given in the following table.

| Large prefix / Symbol | Example | Data type |
|-----------------------|----------------------------------|-----------|
| no prefix | %I10, %CH203.MOD, %CH203.MOD.ERR | BOOL |
| X | %MX20 | BOOL |
| B | %QB102.3 | BYTE |
| W | %KW43 | INT |
| D | %QD100 | DINT |
| F | %MF100 | REAL |

Using Other Data Types

Should other data types be assigned as the default data types of an address, this must be done through an explicit declaration. This variable declaration takes place comfortably using the variable editor. The data type of an address can not be declared directly in an ST section (e.g. declaration `AT %MW1 : UINT`; not permitted).

The following variables are declared in the variable editor:

```
UnlocV1: ARRAY [1..10] OF INT;  
LocV1:   ARRAY [1..10] OF INT AT %MW100;  
LocV2:   TIME AT %MW100;
```

The following calls then have the correct syntax:

```
%MW200 := 5;  
LD LocV1[%MW200]  
ST UnlocV1[2]
```

```
LD t#3s  
ST LocV2
```

Accessing Field Variables

When accessing field variables (`ARRAY`), only literals and variables of `INT`, `DINT`, `UINT` and `UDINT` types are permitted in the index entry.

The index of an `ARRAY` element can be negative if the lower threshold of the range is negative.

Example: Saving a field variable

```
LD var1[i]  
ST var2.otto[4]
```

Modifier

Introduction

Modifiers influence the execution of the operators (see [Operators](#), page 381).

Table of Modifiers

Table of Modifiers:

| Modifier | Use of Operators of data type | Description |
|----------|-------------------------------|--|
| N | BOOL, BYTE, WORD, DWORD | <p>The modifier N is used to invert the value of the operands bit by bit.</p> <p>Example: In the example C is 1, if A is 1 and B is 0.</p> <pre>LD A ANDN B ST C</pre> |
| C | BOOL | <p>The modifier C is used to carry out the associated instruction, should the value of the accumulator be 1 (TRUE).</p> <p>Example: In the example, the jump after START is only performed when A is 1 (TRUE) and B is 1 (TRUE).</p> <pre>LD A AND B JMPC START</pre> |

| Modifier | Use of Operators of data type | Description |
|----------|-------------------------------|---|
| CN | BOOL | <p>If the modifiers C and N are combined, the associated instruction is only performed if the value of the accumulator be a Boolean 0 (FALSE).</p> <p>Example: In the example, the jump after START is only performed when A is 0 (FALSE) and B is 0 (FALSE).</p> <pre>LD A AND B JMPCN START</pre> |
| (| all | <p>The left bracket modifier (is used to move back the evaluation of the operand, until the right bracket operator) appears. The number of right parenthesis operations must be equal to the number of left bracket modifiers. Brackets can be nested.</p> <p>Example: In the example E is 1, if C and/or D is 1 and A and B are 1.</p> <pre>LD A AND B AND (C OR D) ST E</pre> <p>The example can also be programmed in the following manner:</p> <pre>LD A AND B AND (LD C OR D) ST E</pre> |

Operators

Introduction

An operator is a symbol for:

- an arithmetic operation to be executed,
- a logical operation to be executed or
- calling an elementary function block - DFBs or subroutines.

Operators are generic, i.e. they adapt automatically to the data type of the operands.

Load and Save Operators

IL programming language load and save operators:

| Operator | Modifier | Meaning | Operands | Description |
|----------|--|---|--|--|
| LD | N (only for operands of data type BOOL, BYTE, WORD or DWORD) | Loads the operands value into the accumulator | Literal, variable, direct address of any data type | <p>The value of an operand is loaded into the accumulator using LD. The size of the accumulator adapts automatically to the data type of the operand. This also applies to the derived data types.</p> <p>Example: In this example the value of A is loaded into the accumulator, the value of B then added and the result saved in E.</p> <pre>LD A ADD B ST E</pre> |
| ST | N (only for operands of data type BOOL, BYTE, WORD or DWORD) | Saves the accumulator value in the operand | Variable, direct address of any data type | <p>The current value of the accumulator is stored in the operand using ST. The data type of the operand must be the same as the "data type" of the accumulator.</p> <p>Example: In this example the value of A is loaded into the accumulator, the value of B then added and the result saved in E.</p> <pre>LD A ADD B ST E</pre> <p>The "old" result is used in subsequent calculations, depending on whether or not an LD follows an ST.</p> <p>Example: In this example the value of A is loaded into the accumulator, the value of B then added and the result saved in E. The value of B is then subtracted from the value of E (current accumulator content) and the result saved in C.</p> <pre>LD A ADD B ST E SUB B ST C</pre> |

Set and Reset Operators

Set and reset operators of the IL programming language:

| Operator | Modifier | Meaning | Operands | Description |
|----------|----------|--|---|---|
| S | - | Sets the operand to 1, when the accumulator content is 1 | Variable, direct address of <small>BOOL</small> data type | <p>S sets the operand to "1" when the current content of the accumulator is a Boolean 1.</p> <p>Example: In this example the value of A is loaded to the accumulator. If the content of the accumulator (value of A) is 1, then OUT is set to 1.</p> <pre>LD A S OUT</pre> <p>Usually this operator is used together with the reset operator R as a pair.</p> <p>Example: This example shows a RS flip-flop (reset dominant) that is controlled through the two Boolean variables A and C.</p> <pre>LD A S OUT LD C R OUT</pre> |
| R | - | Sets the operand to 0 when the accumulator content is 1 | Variable, direct address of <small>BOOL</small> data type | <p>R sets the operand to "0" when the current content of the accumulator is a Boolean 1.</p> <p>Example: In this example the value of A is loaded to the accumulator. If the content of the accumulator (value of A) is 1, then OUT is set to 0.</p> <pre>LD A R OUT</pre> <p>Usually this operator is used together with the set operator S as a pair.</p> <p>Example: This example shows a SR flip-flop (set dominant) that is controlled through the two Boolean variables A and C.</p> <pre>LD A R OUT LD C S OUT</pre> |

Logical Operators

IL programming language logic operators:

| Operator | Modifier | Meaning | Operands | Description |
|----------|-----------|-------------|---|---|
| AND | N, N (, (| Logical AND | Literal, variable, direct address of BOOL, BYTE, WORD or DWORD data types | <p>The AND operator makes a logical AND link between the accumulator content and the operand.</p> <p>In the case of BYTE, WORD and DWORD data types, the link is made bit by bit.</p> <p>Example: In the example D is 1 if A, B and C are 1.</p> <pre>LD A AND B AND C ST D</pre> |
| OR | N, N (, (| Logical OR | Literal, variable, direct address of BOOL, BYTE, WORD or DWORD data types | <p>The OR operator makes a logical OR link between the accumulator content and the operand.</p> <p>In the case of BYTE, WORD and DWORD data types, the link is made bit by bit.</p> <p>Example: In the example D is 1 if A or B are 1 and C is 1.</p> <pre>LD A OR B OR C ST D</pre> |

| Operator | Modifier | Meaning | Operands | Description |
|----------|-----------|-------------------------------|---|--|
| XOR | N, N (, (| Logical exclusive OR | Literal, variable, direct address of BOOL , BYTE , WORD or DWORD data types | <p>The XOR operator makes a logical exclusive OR link between the accumulator content and the operand.</p> <p>If more than two operands are linked, the result with an uneven number of 1-states is 1, and is 0 with an even number of 1-states.</p> <p>In the case of BYTE, WORD and DWORD data types, the link is made bit by bit.</p> <p>Example: In the example D is 1 if A or B is 1. If A and B have the same status (both 0 or 1), D is 0.</p> <pre>LD A XOR B ST D</pre> <p>If more than two operands are linked, the result with an uneven number of 1-states is 1, and is 0 with an even number of 1-states.</p> <p>Example: In the example F is 1 if 1 or 3 operands are 1. F is 0 if 0, 2 or 4 operands are 1.</p> <pre>LD A XOR B XOR C XOR D XOR E ST F</pre> |
| NOT | - | Logical negation (complement) | Accumulator contents of data types BOOL , BYTE , WORD or DWORD | <p>The accumulator content is inverted bit by bit with NOT.</p> <p>Example: In the example B is 1 if A is 0 and B is 0 if A is 1.</p> <pre>LD A NOT ST B</pre> |

Arithmetic Operators

IL programming language Arithmetic operators:

| Operator | Modifier | Meaning | Operands | Description |
|----------|----------|----------------|--|---|
| ADD | (| Addition | Literal, variable, direct address of data types INT, DINT, UINT, UDINT, REAL or TIME | <p>With ADD the value of the operand is added to the value of the accumulator contents.</p> <p>Example: The example corresponds to the formula $D = A + B + C$</p> <p>LD A</p> <p>ADD B</p> <p>ADD C</p> <p>ST D</p> |
| SUB | (| Subtraction | Literal, variable, direct address of data types INT, DINT, UINT, UDINT, REAL or TIME | <p>With SUB the value of the operand is subtracted from the accumulator content.</p> <p>Example: The example corresponds to the formula $D = A - B - C$</p> <p>LD A</p> <p>SUB B</p> <p>SUB C</p> <p>ST D</p> |
| MUL | (| Multiplication | Literal, variable, direct address of data type INT, DINT, UINT, UDINT or REAL | <p>The MUL operator multiplies the content of the accumulator by the value of the operand.</p> <p>Example: The example corresponds to the formula $D = A * B * C$</p> <p>LD A</p> <p>MUL B</p> <p>MUL C</p> <p>ST D</p> <p>Note: The MULTIME function in the obsolete library is available for multiplications involving the data type Time.</p> |

| Operator | Modifier | Meaning | Operands | Description |
|----------|----------|-----------------|---|--|
| DIV | (| Division | Literal, variable, direct address of data type INT, DINT, UINT, UDINT or REAL | <p>The DIV operator divides the contents of the accumulator by the value of the operand.</p> <p>Example: The example corresponds to the formula $D = A / B / C$</p> <pre>LD A DIV B DIV C ST D</pre> <p>Note: The DIVTIME function in the obsolete library is available for divisions involving the data type Time.</p> |
| MOD | (| Modulo Division | Literal, variable, direct address of INT, DINT, UINT or UDINT data types | <p>The MOD operator divides the value of the first operand by the value of the second and returns the remainder (Modulo) as the result.</p> <p>Example: In this example</p> <ul style="list-style-type: none"> • C is 1 if A is 7 and B is 2 • C is 1 if A is 7 and B is -2 • C is -1 if A is -7 and B is 2 • C is -1 if A is -7 and B is -2 <pre>LD A MOD B ST C</pre> |

Comparison Operators

IL programming language comparison operators:

| Operator | Modifier | Meaning | Operands | Description |
|----------|----------|-------------------|---|--|
| GT | (| Comparison: > | Literal, variable, direct address of data type <code>BOOL</code> , <code>BYTE</code> , <code>WORD</code> , <code>DWORD</code> , <code>STRING</code> , <code>INT</code> , <code>DINT</code> , <code>UINT</code> , <code>UDINT</code> , <code>REAL</code> , <code>TIME</code> , <code>DATE</code> , <code>DT</code> or <code>TOD</code> | <p>The <code>GT</code> operator compares the contents of the accumulator with the contents of the operand. If the contents of the accumulator are greater than the contents of the operands, the result is a Boolean 1. If the contents of the accumulator are less than/equal to contents of the operands, the result is a Boolean 0.</p> <p>Example: In the example the value of <code>D</code> is 1 if <code>A</code> is greater than 10, otherwise the value of <code>D</code> is 0.</p> <pre>LD A GT 10 ST D</pre> |
| GE | (| Comparison: >= | Literal, variable, direct address of data type <code>BOOL</code> , <code>BYTE</code> , <code>WORD</code> , <code>DWORD</code> , <code>STRING</code> , <code>INT</code> , <code>DINT</code> , <code>UINT</code> , <code>UDINT</code> , <code>REAL</code> , <code>TIME</code> , <code>DATE</code> , <code>DT</code> or <code>TOD</code> | <p>The <code>GE</code> operator compares the contents of the accumulator with the contents of the operand. If the contents of the accumulator are greater than/equal to the contents of the operands, the result is a Boolean 1. If the contents of the accumulator are less than the contents of the operands, the result is a Boolean 0.</p> <p>Example: In the example the value of <code>D</code> is 1 if <code>A</code> is greater than or equal to 10, otherwise the value of <code>D</code> is 0.</p> <pre>LD A GE 10 ST D</pre> |
| EQ | (| Comparison: = | Literal, variable, direct address of data type <code>BOOL</code> , <code>BYTE</code> , <code>WORD</code> , <code>DWORD</code> , <code>STRING</code> , <code>INT</code> , <code>DINT</code> , <code>UINT</code> , <code>UDINT</code> , <code>REAL</code> , <code>TIME</code> , <code>DATE</code> , <code>DT</code> or <code>TOD</code> | <p>The <code>EQ</code> operator compares the contents of the accumulator with the contents of the operand. If the contents of the accumulator is equal to the contents of the operands, the result is a Boolean 1. If the contents of the accumulator are not equal to the contents of the operands, the result is a Boolean 0.</p> <p>Example: In the example the value of <code>D</code> is 1 if <code>A</code> is equal to 10, otherwise the value of <code>D</code> is 0.</p> <pre>LD A EQ 10 ST D</pre> |
| NE | (| Comparison: <> | Literal, variable, direct address of data type <code>BOOL</code> , <code>BYTE</code> , <code>WORD</code> , <code>DWORD</code> , <code>STRING</code> , <code>INT</code> , <code>DINT</code> , <code>UINT</code> , <code>UDINT</code> , <code>REAL</code> , | <p>The <code>NE</code> operator compares the contents of the accumulator with the contents of the operand. If the contents of the accumulator are not equal to the contents of the operands, the result is a Boolean 1. If the contents of the accumulator are equal to the contents of the operands, the result is a Boolean 0.</p> |

| Operator | Modifier | Meaning | Operands | Description |
|----------|----------|-------------------|---|--|
| | | | TIME, DATE, DT or TOD | <p>Example: In the example the value of D is 1 if A is not equal to 10, otherwise the value of D is 0.</p> <p>LD A</p> <p>NE 10</p> <p>ST D</p> |
| LE | (| Comparison: <= | Literal, variable, direct address of data type BOOL, BYTE, WORD, DWORD, STRING, INT, DINT, UINT, UDINT, REAL, TIME, DATE, DT or TOD | <p>The LE operator compares the contents of the accumulator with the contents of the operand. If the contents of the accumulator are less than/ equal to the contents of the operands, the result is a Boolean 1. If the contents of the accumulator are greater than the contents of the operands, the result is a Boolean 0.</p> <p>Example: In the example the value of D is 1 if A is smaller than or equal to 10, otherwise the value of D is 0.</p> <p>LD A</p> <p>LE 10</p> <p>ST D</p> |
| LT | (| Comparison: < | Literal, variable, direct address of data type BOOL, BYTE, WORD, DWORD, STRING, INT, DINT, UINT, UDINT, REAL, TIME, DATE, DT or TOD | <p>The LT operator compares the contents of the accumulator with the contents of the operand. If the contents of the accumulator is less than the contents of the operands, the result is a Boolean 1. If the contents of the accumulator is greater than/ equal to contents of the operands, the result is a Boolean 0.</p> <p>Example: In the example the value of D is 1 if A is smaller than 10, otherwise the value of D is 0.</p> <p>LD A</p> <p>LT 10</p> <p>ST D</p> |

Call Operators

IL programming language call operators:

| Operator | Modifier | Meaning | Operands | Description |
|----------------|--|---|---|--|
| CAL | C, CN (only if the accumulator contents are of the <code>BOOL</code> data type) | Call of a function block, DFB or subprogram | Instance name of the function block, DFB or subprogram | A function block, DFB or subprogram is called up conditionally or unconditionally with <code>CAL</code> . see also Calling Elementary Function Blocks and Derived Function Blocks, page 401 and Subroutine Call, page 393 |
| FUNCTION-NAME | - | Executing a function | Literal, variable, direct address (data type is dependent on function) | A function is performed by specifying the name of the function. see also Calling Elementary Functions, page 395 |
| PROCEDURE-NAME | - | Executing a procedure | Literal, variable, direct address (data type is dependent on procedure) | A procedure is performed by specifying the name of the procedure. see also Calling Procedures, page 412 |

Structuring Operators

IL programming language structuring operators:

| Operator | Modifier | Meaning | Operands | Description |
|----------|--|--|----------|---|
| JMP | C, CN (only if the accumulator contents are of the <code>BOOL</code> data type) | Jump to label | LABEL | With <code>JMP</code> a jump to the label can be conditional or unconditional. see also <code>Labels and Jumps</code> , page 393 |
| RET | C, CN (only if the accumulator contents are of the <code>BOOL</code> data type) | Return to the next highest program organization unit | - | <code>RETURN</code> operators can be used in DFBs (derived function blocks) and in SRs (subroutines). <code>RETURN</code> operators can not be used in the main program. <ul style="list-style-type: none"> In a DFB, a <code>RETURN</code> operator forces the return to the program which called the DFB. <ul style="list-style-type: none"> The rest of the DFB section containing the <code>RETURN</code> operator is not executed. The next sections of the DFB are not executed. The program which called the DFB will be executed after return from the DFB. If the DFB is called by another DFB, the calling DFB will be executed after return. In a SR, a <code>RETURN</code> operator forces the return to the program which called the SR. <ul style="list-style-type: none"> The rest of the SR containing the <code>RETURN</code> operator is not executed. The program which called the SR will be executed after return from the SR. |
|) | - | Processing deferred operations | - | A right bracket <code>)</code> starts the processing of the deferred operator. The number of right bracket operations must be equal to the number of left bracket modifiers. Brackets can be nested. Example: In the example <code>E</code> is 1 if <code>C</code> and/or <code>D</code> is 1 and <code>A</code> and <code>B</code> are 1. <pre>LD A AND B AND (C OR D) ST E</pre> |

Subroutine Call

Call Subroutine

A subroutine call consists of the `CAL` operator, followed by the name of the subroutine section, followed by an empty parameter list (optional).

Subroutine calls do not return a value.

The subroutine to be called must be located in the same task as the IL section called.

Subroutines can also be called from within subroutines.

e.g.

```
ST A
CAL SubroutineName ()
LD B
```

or

```
ST A
CAL SubroutineName
LD B
```

Subroutines are a supplement to IEC 61131-3 and must be enabled explicitly.

In SFC action sections, subroutine calls are only allowed when Multitoken Operation is enabled.

Labels and Jumps

Introduction

Labels serve as destinations for Jumps.

Label Properties:

Label properties:

- Labels must always be the first element in a line.
- The name must be clear throughout the directory, and it is not upper/lower case sensitive.
- Labels can be 256 characters long (max.).
- Labels must conform to the IEC name conventions.
- Labels are separated by a colon `:` from the following instruction.

- Labels are only permitted at the beginning of "Expressions", otherwise an undefined value can be found in the battery.

Example:

```
start: LD A
      AND B
      OR C
      ST D
      JMP start
```

Jump Properties:

Jump properties:

- With `JMP` operation a jump to the label can be restricted or unrestricted.
- `JMP` can be used with the modifiers `C` and `CN` (only if the battery content is data type `BOOL`).
- Jumps can be made within program and DFB sections.
- Jumps are only possible in the current section.

Possible destinations are:

- the first `LD` instruction of an EFB/DFB call with assignment of input parameters (see `start2`),
- a normal `LD` instruction (see `start1`),
- a `CAL` instruction, which does not work with assignment of input parameters (see `start3`),
- a `JMP` instruction (see `start4`),
- the end of an instruction list (see `start5`).

Example

```
start2: LD A
      ST counter.CU
      LD B
      ST counter.R
      LD C
      ST counter.PV
      CAL counter
      JMPCN start4
start1: LD A
      AND B
      OR C
      ST D
      JMPC start3
```

```
LD A
ADD E
JMP start5
start3: CAL counter (
CU:=A
R:=B
PV:=C )
JMP start1
LD A
OR B
OR C
ST D
start4: JMPC start1
LD C
OR B
start5: ST A
```

Comment

Description

In the IL editor, comments always start with the string (* and end in the string *). Any comments can be entered between these character strings.

Nesting comments is not permitted according to IEC 61131-3. If comments are nested nevertheless, then they must be enabled explicitly.

Calling Elementary Functions, Elementary Function Blocks, Derived Function Blocks and Procedures

Overview

Calling Elementary Functions, Elementary Function Blocks, Derived Function Blocks and Procedures in the IL programming language.

Calling Elementary Functions

Using Functions

Elementary functions are provided in the form of libraries. The logic of the functions is created in the programming language C and may not be modified in the IL editor.

Functions have no internal states. If the input values are the same, the value on the output is the same every time the function is called. For example, the addition of two values always gives the same result. With some elementary functions, the number of inputs can be increased.

Elementary functions only have one return value (output).

Parameters

"Inputs" and one "output" are required to transfer values to or from a function. These are called formal parameters.

The current process states are transferred to the formal parameters. These are called actual parameters.

The following can be used as actual parameters for function inputs:

- Variable
- Address
- Literal

The following can be used as actual parameters for function outputs:

- Variable
- Address

The data type of the actual parameters must match the data type of the formal parameters. The only exceptions are generic formal parameters whose data type is determined by the actual parameter.

When dealing with generic `ANY_BIT` formal parameters, actual parameters of the `INT` or `DINT` (not `UINT` and `UDINT`) data types can be used.

This is a supplement to IEC 61131-3 and must be enabled explicitly.

Example:

Allowed:

```
AND (AnyBitParam := IntVar1, AnyBitParam2 := IntVar2)
```

Not allowed:

```
AND_WORD (WordParam1 := IntVar1, WordParam2 := IntVar2)
```

(In this case, `AND_INT` must be used.)

```
AND_ARRAY_WORD (ArrayInt, ...)
```

(In this case an explicit type conversion must be carried out using `INT_ARR_TO_WORD_ARR (...)`).

Not all formal parameters must be assigned a value for formal calls. The formal parameter types that must be assigned a value are in the following table:

| Parameter type | EDT | STRING | ARRAY | ANY_ARRAY | IODDT | STRUC-T | FB | ANY |
|--|-----|--------|-------|-----------|-------|---------|----|-----|
| Input | - | - | - | - | + | - | + | - |
| VAR_IN_OUT | + | + | + | + | + | + | / | + |
| Output | - | - | - | - | - | - | / | - |
| + Actual parameter required | | | | | | | | |
| - Actual parameter not required, it's the general rule, but there are exceptions for some FFBs, for instance when some parameters are used to characterize the information we want to be given by the FFB. | | | | | | | | |
| / not applicable | | | | | | | | |

If no value is assigned to a formal parameter, the initial value will be used when the function is executed. If no initial value has been defined, the default value (0) is used.

Programming Notes

Attention should be paid to the following programming notes:

- Functions are only executed if the input EN=1 or the EN input is not used (see also EN and ENO, page 400).
- All generic functions are overloaded. This means the functions can be called with or without entering the data type.

E.g.

```
LD i1
ADD i2
ST i3
```

is identical to

```
LD i1
ADD_INT i2
ST i3
```

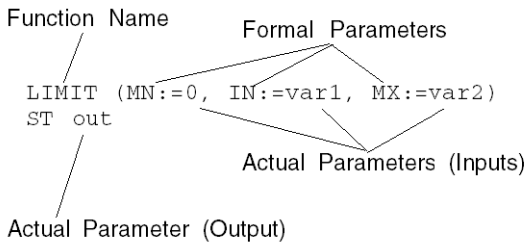
- In contrast to ST, functions in IL cannot be nested.
- There are two ways of invoking a function:
 - Formal call (calling a function with formal parameter names)
 - Informal call (calling a function without formal parameter names)

Formal Call

With this type of call (call with formal parameter names), the function is called using an instruction sequence consisting of the function name, followed by the bracketed list of value assignments (actual parameters) to the formal parameters. The order in which the formal parameters are listed is **not significant**. The list of actual parameters may be wrapped immediately following a comma. After executing the function the result is loaded into the accumulator and can be stored using `ST`.

`EN` and `ENO` can be used for this type of call.

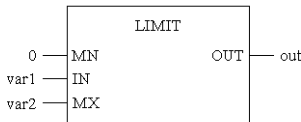
Calling a function with formal parameter names:



or

```
LIMIT (
MN:=0,
IN:=var1,
MX:=var2
)
ST out
```

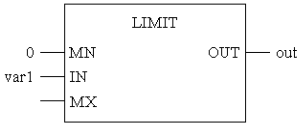
Calling the same function in FBD:



With formal calls, values do not have to be assigned to all formal parameters (see also Parameter, page 396).

```
LIMIT (MN:=0, IN:=var1)
ST out
```

Calling the same function in FBD:

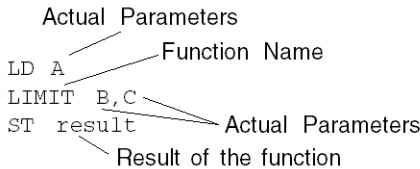


Informal Call

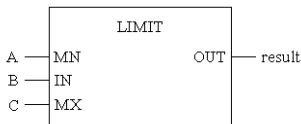
With this type of call (call without formal parameter names), the function is called using an instruction sequence made up by loading the first actual parameter into the accumulator, followed by the function name and an optional list of actual parameters. The order in which the actual parameters are listed is **significant**. The list of actual parameters cannot be wrapped. After executing the function the result is loaded into the accumulator and can be stored using `ST`.

`EN` and `ENO` **cannot** be used for this type of call.

Calling a function with formal parameter names:



Calling the same function in FBD:



NOTE: Note that when making an informal call, the list of actual parameters **cannot** be put in brackets. IEC 61133-3 requires that the brackets be left out in this case to illustrate that the first actual parameter is not a part of the list.

Invalid informal call for a function:

```
LD A
LIMIT (B,C)
```

If the value to be processed (first actual parameter) is already in the accumulator, the load instruction can be omitted.

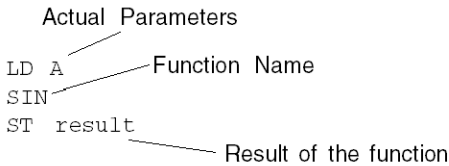
```
LIMIT B,C
ST result
```

If the result is to be used immediately, the store instruction can be omitted.

```
LD A
LIMIT_REAL B,C
MUL E
```

If the function to be executed only has one input, the name of the function is not followed by a list of actual parameters.

Calling a function with one actual parameter:



Calling the same function in FBD:



EN and ENO

With all functions an EN input and an ENO output can be configured.

If the value of EN is equal to "0" when the function is called, the algorithms defined by the function are not executed and ENO is set to "0".

If the value of EN is equal to 1 when the function is called, the algorithms defined by the function are executed. After the algorithms have been executed successfully, the value of ENO is set to "1". If an error occurred while executing the algorithms, ENO is set to "0".

If the EN pin is not assigned a value, when the FFB is invoked, the algorithm defined by the FFB is executed (same as if EN equals to "1").

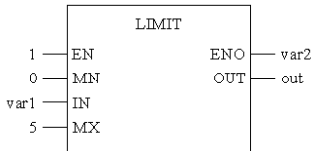
If ENO is set to "0" (caused when EN=0 or an error occurred during execution), the output of the function is set to "0".

The output behavior of the function does not depend on whether the function was called up without EN/ENO or with EN=1.

If EN/ENO are used, the function call must be formal.

```
LIMIT (EN:=1, MN:=0, IN:=var1, MX:=5, ENO=>var2)
ST out
```


Calling the same function in FBD:



Calling Elementary Function Blocks and Derived Function Blocks

Elementary Function Block

Elementary function blocks have internal states. If the inputs have the same values, the value on the output can have another value during the individual operations. For example, with a counter, the value on the output is incremented.

Function blocks can have several output values (outputs).

Derived Function Block

Derived function blocks (DFBs) have the same properties as elementary function blocks. The user can create them in the programming languages FBD, LD, IL, and/or ST.

Parameter

"Inputs and outputs" are required to transfer values to or from function blocks. These are called formal parameters.

The current process states are transferred to the formal parameters. They are called actual parameters.

The following can be used as actual parameters for function block inputs:

- Variable
- Address
- Literal

The following can be used as actual parameters for function block outputs:

- Variable
- Address

The data type of the actual parameters must match the data type of the formal parameters. The only exceptions are generic formal parameters whose data type is determined by the actual parameter.

Exception:

When dealing with generic ANY_BIT formal parameters, actual INT or DINT (not UINT and UDINT) parameters can be used.

This is a supplement to IEC 61131-3 and must be enabled explicitly.

Example:

Allowed:

```
AND (AnyBitParam := IntVar1, AnyBitParam2 := IntVar2)
```

Not allowed:

```
AND_WORD (WordParam1 := IntVar1, WordParam2 := IntVar2)
```

(In this case, AND_INT must be used.)

```
AND_ARRAY_WORD (ArrayInt, ...)
```

(In this case an explicit type conversion must be carried out using INT_ARR_TO_WORD_ARR (...)).

Not all formal parameters need be assigned a value. The formal parameter types that must be assigned a value are in the following table:

| Parameter type | EDT | STRING | ARRAY | ANY_ARRAY | IODDT | DEVICE DDT | FB | ANY |
|--|-----|--------|-------|-----------|-------|------------|----|-----|
| EFB: Input | - | - | - | - | / | / | / | - |
| EFB: VAR_IN_OUT | + | + | + | + | + | / | / | + |
| EFB: Output | - | - | + | + | + | / | / | + |
| DFB: Input | - | - | - | - | / | + | / | - |
| DFB: VAR_IN_OUT | + | + | + | + | + | + | / | + |
| DFB: Output | - | - | + | / | / | / | / | + |
| + Actual parameter required | | | | | | | | |
| - Actual parameter not required, it's the general rule, but there are exceptions for some FFBs, for instance when some parameters are used to characterize the information we want to be given by the FFB. | | | | | | | | |
| / not applicable | | | | | | | | |

If no value is allocated to a formal parameter, then the initial value is used for executing the function block. If no initial value has been defined then the default value (0) is used.

If a formal parameter is not assigned a value and the function block/DFB is instanced more than once, then the following instances are run with the old value.

NOTE: An ANY_ARRAY_xxx input pin not connected will create automatically an hidden array of 1 element.

Public Variables

In addition to inputs and outputs, some function blocks also provide public variables.

These variables transfer statistical values (values that are not influenced by the process) to the function block. They are used for setting parameters for the function block.

Public variables are a supplement to IEC 61131-3.

The assignment of values to public variables is made via their initial values or via the load and save instructions.

Example:

| | | | |
|---------------------------|---------------|-----------------|---|
| | Instance Name | Public Variable | |
| LD 1 ST D_ACT1.OP_CTRL | / | / | (D_ACT1 is an instance of the function block D_ACT and has the public variables AREA_NR and OP_CTRL.) |

Public variables are read via the instance name of the function block and the names of the public variables.

Example:

| | | |
|------------------------------|---------------|-----------------|
| | Instance Name | Public Variable |
| LD D_ACT1.OP_CTRL ST Var1 | / | / |

Private Variables

In addition to inputs, outputs and public variables, some function blocks also provide private variables.

Like public variables, private variables are used to transfer statistical values (values that are not influenced by the process) to the function block.

Private variables can not be accessed by user program. These type of variables can only be accessed by the animation table.

NOTE: Nested DFBs are declared as private variables of the parent DFB. So their variables are also not accessible through programming, but through the animation table. Private variables are a supplement to IEC 61131-3.

Programming Notes

Attention should be paid to the following programming notes:

- Functions are only executed if the input `EN=1` or the `EN` input is not used (see also `EN` and `ENO`, page 410).
- The assignment of variables to `ANY` or `ARRAY` output types must be made using the `=>` operator (see also *Formal Form of CAL with a List of the Input Parameters*, page 405).

Assignments cannot be made outside the function block call.

The instruction

```
My_Var := My_SAH.OUT
```

is **invalid**, if the output `OUT` of the `SAH` function block is of type `ANY`.

The instruction

```
Cal My_SAH (OUT=>My_Var)
```

is **valid**.

- Special conditions apply when using `VAR_IN_OUT` variables, page 411.
- The use of function blocks consists of two parts:
 - the Declaration, page 405
 - calling the function block
- There are four ways of calling a function block:
 - Formal Form of CAL with a list of input parameters, page 405 (call with formal parameter names)
In this case variables can be assigned to outputs using the `=>` operator.
 - Informal form of CAL with a list of input parameters, page 406 (call without formal parameter names)
 - CAL and Load/Save, page 407 the input parameter
 - Use of the input operators, page 408
- Function block/DFB instances can be called multiple times; other than instances of communication EFBs, these can only be called once (see *Multiple Call of a Function Block Instance*, page 410).

Declaration

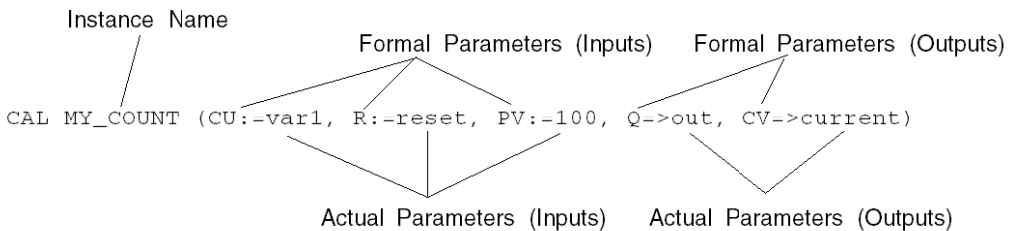
Before calling a function block it must be declared in the variables editor.

Formal Form of CAL with a List of Input Parameters

With this type of call (call with formal parameter names), the function block is called using a CAL instruction which follows the instance name of the function block and a bracketed list of actual parameter assignments to the formal parameters. The assignment of the input formal parameter is made using the := assignment and the output formal parameter is made using the => assignment. The sequence in which the input formal parameters and output formal parameters are enumerated is **not significant**. The list of actual parameters may be continued immediately following a comma.

EN and ENO can be used for this type of call.

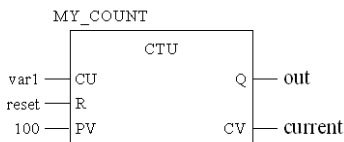
Function block call in the formal form of CAL with a list of input parameters:



or

```
CAL MY_COUNT (CU:=var1,
              R:=reset,
              PV:=100,
              Q=>out,
              CV=>current)
```

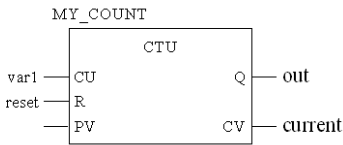
Calling the same function block in FBD:



It is not necessary to assign a value to all formal parameters (see also Parameter, page 401).

```
CAL MY_COUNT (CU:=var1, R:=reset, Q=>out, CV=>current)
```

Calling the same function block in FBD:



The value of a function block output can be stored and then saved by loading the function block output (function block instance name and separated by a full stop or entering the formal parameter).

Loading and saving function block outputs:

```

Instance Name Formal Parameter (Output)
LD MY_COUNT.Q
ST out Actual Parameter (Output)
LD MY_COUNT.CV
ST current

```

Informal Form of CAL with a List of Input Parameters

With this type of call (call without formal parameter names), the function block is called using a CAL instruction, that follows the instance name of the function block and a bracketed list of actual parameter for the inputs and outputs. The order in which the actual parameters are listed in a function block call **is significant**. The list of actual parameters cannot be wrapped.

EN and ENO **cannot** be used for this type of call.

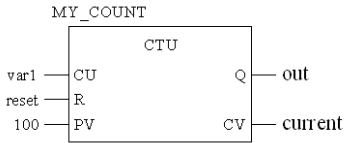
Function block call in the informal form of CAL with a list of input parameters:

```

Instance Name Actual Parameter (Input)
CAL MY_COUNT (var1, reset, 100, out, current)
Actual Parameter (Output)

```

Calling the same function block in FBD:



With informal calls it is not necessary to assign a value to all formal parameters (see also Parameter, page 401).

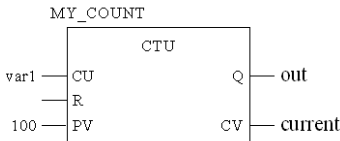
This is a supplement to IEC 61131-3 and must be enabled explicitly.

An empty parameter field is used to omit a parameter.

Call with empty parameter field:

```
CAL MY_COUNT (var1, , 100, out, current)
```

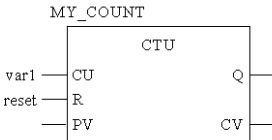
Calling the same function block in FBD:



An empty parameter field does not have to be used if formal parameters are omitted at the end.

```
MY_COUNT (var1, reset)
```

Calling the same function block in FBD:



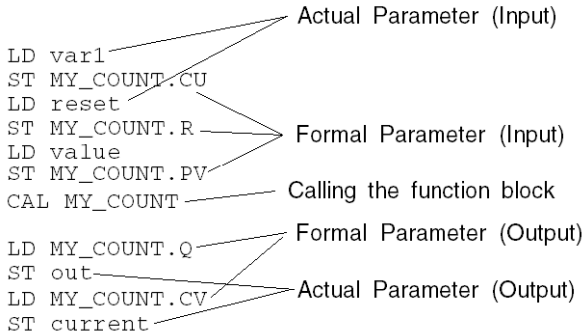
CAL and Load/Save the Input Parameters

Function blocks may be called with an instruction list consisting of loading the actual parameters, followed by saving into the formal parameter, followed by the CAL instruction. The sequence of loading and saving the parameters is **not significant**.

Only load and save instructions for the function block currently being configured are allowed between the first load instruction for the actual parameters and the call of the function block. All other instructions are not allowed in this position.

It is not necessary to assign a value to all formal parameters (see also Parameter, page 401).

CAL with Load/Save the input parameters:



Use of the Input Operators

Function blocks can be called using an instruction list that consists of loading the actual parameters followed by saving them in the formal parameters followed by an input operator. The sequence of loading and saving the parameters is **not significant**.

Only load and save instructions for the function block currently being configured are allowed between the first load instruction for the actual parameters and the input operator of the function block. All other instructions are not allowed in this position.

EN and ENO **cannot** be used for this type of call.

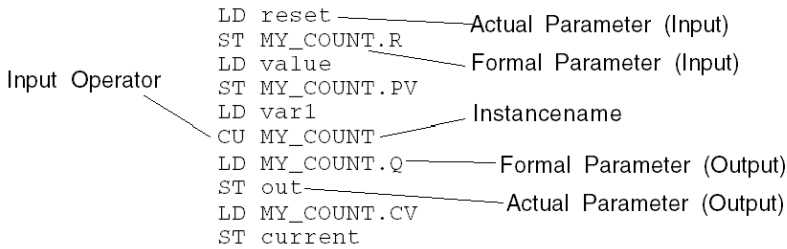
It is not necessary to assign a value to all formal parameters (see also Parameter, page 401).

The possible input operators for the various function blocks can be found in the table. Additional input operators are not available.

| Input Operator | FB type |
|-------------------|--|
| S1, R | SR |
| S, R1 | RS |
| CLK | R_TRIG |
| CLK | F_TRIG |
| CU, R, PV | CTU_INT, CTU_DINT, CTU_UINT, CTU_UDINT |
| CD, LD, PV | CTD_INT, CTD_DINT, CTD_UINT, CTD_UDINT |
| CU, CD, R, LD, PV | CTUD_INT, CTUD_DINT, CTUD_UINT, CTUD_UDINT |

| Input Operator | FB type |
|----------------|---------|
| IN, PT | TP |
| IN, PT | TON |
| IN, PT | TOF |

Use of the input operators:



Calling a Function Block without Inputs

Even if the function block has no inputs or the inputs are not to be parameterized, the function block should be called before its outputs can be used. Otherwise the initial values of the outputs will be transferred, i.e. "0".

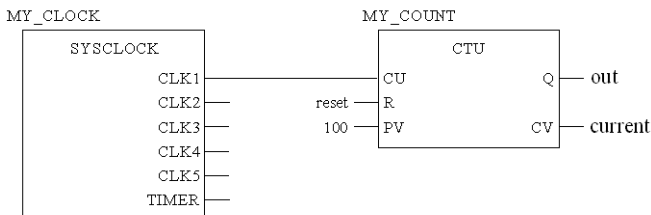
E.g.

Calling the function block in the IL programming language:

```

CAL MY_CLOCK () CAL MY_COUNT (CU:=MY_CLOCK.CLK1, R:=reset, PV:=100)
LD MY_COUNT.Q
ST out
LD MY_COUNT.CV
ST current
    
```

Calling the same function block in FBD:



Multiple Function Block Instance Call

Function block/DFB instances can be called multiple times; other than instances of communication EFBs, these can only be called once.

Calling the same function block/DFB instance more than once makes sense, for example, in the following cases:

- If the function block/DFB has no internal value or it is not required for further processing.

In this case, memory is saved by calling the same function block/DFB instance more than once since the code for the function block/DFB is only loaded one time.

The function block/DFB is then handled like a "Function".

- If the function block/DFB has an internal value and this is supposed to influence various program segments, for example, the value of a counter should be increased in different parts of the program.

In this case, calling the same function block/DFB means that temporary results do not have to be saved for further processing in another part of the program.

EN and ENO

With all function blocks/DFBs, an `EN` input and an `ENO` output can be configured.

If the value of `EN` is equal to "0", when the function block/DFB is called, the algorithms defined by the function block/DFB are not executed and `ENO` is set to "0".

If the value of `EN` is equal to "1", when the function block/DFB is invoked, the algorithms which are defined by the function block/DFB will be executed. After the algorithms have been executed successfully, the value of `ENO` is set to "1". If an error occurs when executing these algorithms, `ENO` is set to "0".

If the `EN` pin is not assigned a value, when the FFB is invoked, the algorithm defined by the FFB is executed (same as if `EN` equals to "1").

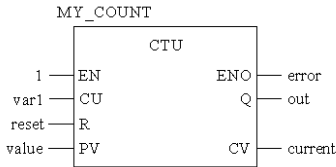
If `ENO` is set to "0" (results from `EN=0` or an error during execution), the outputs of the function block/DFB retain the status from the last cycle in which they were correctly executed.

The output behavior of the function blocks/DFBs does not depend on whether the function blocks/DFBs are called without `EN/ENO` or with `EN=1`.

If `EN/ENO` are used, the function block call must be formal. The assignment of variables to `ENO` must be made using the `=>` operator.

```
CAL MY_COUNT (EN:=1, CU:=var1, R:=reset, PV:=value,
              ENO=>error, Q=>out, CV=>current) ;
```

Calling the same function block in FBD:



VAR_IN_OUT Variable

Function blocks are often used to read a variable at an input (input variables), to process it and to output the updated values of the same variable (output variables). This special type of input/output variable is also called a VAR_IN_OUT variable.

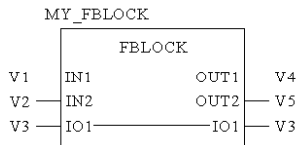
The following special features are to be noted when using function blocks/DFBs with VAR_IN_OUT variables:

- All VAR_IN_OUT inputs must be assigned a variable.
- VAR_IN_OUT inputs may not have literals or constants assigned to them.
- VAR_IN_OUT outputs may **not** have values assigned to them.
- VAR_IN_OUT variables **cannot** be used outside the block call.

Calling a function block with a VAR_IN_OUT variable in IL:

```
CAL MY_FBLOCK(IN1:=V1, IN2:=V2, IO1:=V3,
              OUT1=>V4, OUT2=>V5)
```

Calling the same function block in FBD:



VAR_IN_OUT variables **cannot** be used outside the function block call.

The following function block calls are therefore **invalid**:

Invalid call, example 1:

| | |
|-------------|--|
| LD V1 | Loading a V1 variable in the accumulator |
| CAL InOutFB | Calling a function block with the VAR_IN_OUT parameter. The accumulator now contains a reference to a VAR_IN_OUT parameter. |
| AND V2 | AND operation on accumulator contents and V2 variable. Error: The operation cannot be performed since the VAR_IN_OUT parameter (accumulator contents) cannot be accessed from outside the function block call. |

Invalid call, example 2:

| | |
|-------------------|---|
| LD V1 | Loading a V1 variable in the accumulator |
| AND InOutFB.inout | AND operation on accumulator contents and a reference to a VAR_IN_OUT parameter. Error: The operation cannot be performed since the VAR_IN_OUT parameter cannot be accessed from outside the function block call. |

The following function block calls are always **valid**:

Valid call, example 1:

| | |
|--------------------------------|---|
| CAL InOutFB (IN1:=V1,inout:=V2 | Calling a function block with the VAR_IN_OUT parameter and assigning the actual parameter within the function block call. |
|--------------------------------|---|

Valid call, example 2:

| | |
|-------------------------|--|
| LD V1 | Loading a V1 variable in the accumulator |
| ST InOutFB.IN1 | Assigning the accumulator contents to the IN1 parameter of the IN1 function block. |
| CAL InOutFB (inout:=V2) | Calling the function block with assignment of the actual parameter (V2) to the VAR_IN_OUT parameter. |

Calling Procedures

Procedure

Procedures are provided in the form of libraries. The logic of the procedure is created in the programming language C and may not be modified in the IL editor.

Procedures - like functions - have no internal states. If the input values are the same, the value on the output is the same every time the procedure is executed. For example, the addition of two values gives the same result every time.

In contrast to functions, procedures do not return a value and support `VAR_IN_OUT` variables.

Procedures are a supplement to IEC 61131-3 and must be enabled explicitly.

Parameter

"Inputs and outputs" are required to transfer values to or from procedures. These are called formal parameters.

The current process states are transferred to the formal parameters. These are called actual parameters.

The following can be used as actual parameters for procedure inputs:

- Variable
- Address
- Literal

The following can be used as actual parameters for procedure outputs:

- Variable
- Address

The data type of the actual parameter must match the data type of the formal parameter. The only exceptions are generic formal parameters whose data type is determined by the actual parameter.

When dealing with generic `ANY_BIT` formal parameters, actual parameters of the `INT` or `DINT` (not `UINT` and `UDINT`) data types can be used.

This is a supplement to IEC 61131-3 and must be enabled explicitly.

Example:

Allowed:

```
AND (AnyBitParam := IntVar1, AnyBitParam2 := IntVar2)
```

Not allowed:

```
AND_WORD (WordParam1 := IntVar1, WordParam2 := IntVar2)
```

(In this case, `AND_INT` must be used.)

```
AND_ARRAY_WORD (ArrayInt, ...)
```

(In this case an explicit type conversion must be carried out using `INT_ARR_TO_WORD_ARR` (...)).

Not all formal parameters must be assigned a value for formal calls. Which formal parameter types must be assigned a value can be seen in the following table.

| Parameter type | EDT | STRING | ARRAY | ANY_ARRAY | IODDT | STRUC-T | FB | ANY |
|---------------------------------|-----|--------|-------|-----------|-------|---------|----|-----|
| Input | - | - | + | + | + | + | + | + |
| VAR_IN_OUT | + | + | + | + | + | + | / | + |
| Output | - | - | - | - | - | - | / | + |
| + Actual parameter required | | | | | | | | |
| - Actual parameter not required | | | | | | | | |
| / not applicable | | | | | | | | |

If no value is allocated to a formal parameter, then the initial value will be used for executing the function block. If no initial value has been defined, the default value (0) is used.

Programming Notes

Attention should be paid to the following programming notes:

- Procedures are only executed if the input EN=1 or the EN input is not used (see also EN and ENO, page 417).
- Special conditions apply when using VAR_IN_OUT variables, page 418.
- There are two ways of calling a procedure:
 - Formal call (calling a function with formal parameter names)
 - In this case variables can be assigned to outputs using the => operator (calling a function block in shortened form).
 - Informal call (calling a function without formal parameter names)

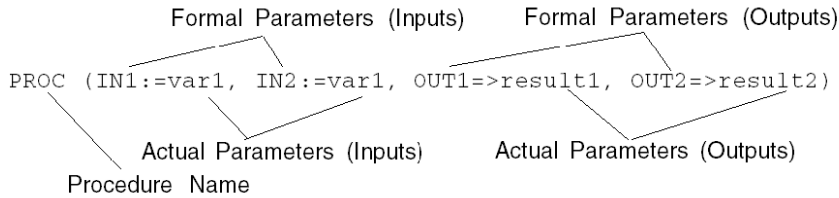
Formal Call

With this type of call (call with formal parameter names), the procedure is called using an optional CAL instruction sequence followed by the name of the procedure and a bracketed list of actual parameter to formal parameter assignments. The assignment of the input formal parameter is made using the := assignment and the output formal parameter is made using the => assignment. The order in which the input formal parameters and output formal parameters are listed is **not significant**.

The list of actual parameters may be wrapped immediately following a comma.

EN and ENO can be used for this type of call.

Calling a procedure with formal parameter names:



or

```
CAL PROC (IN1:=var1, IN2:=var1, OUT1=>result1,OUT2=>result2)
```

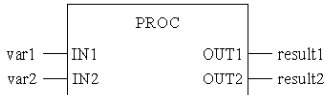
or

```
PROC (IN1:=var1,
      IN2:=var1,
      OUT1=>result1,
      OUT2=>result2)
```

or

```
CAL PROC (IN1:=var1,
          IN2:=var1,
          OUT1=>result1,
          OUT2=>result2)
```

Calling the same procedure in FBD:



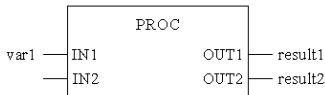
With formal calls, values do not have to be assigned to all formal parameters (see also Parameter, page 413).

```
PROC (IN1:=var1, OUT1=>result1, OUT2=>result2)
```

or

```
CAL PROC (IN1:=var1, OUT1=>result1, OUT2=>result2)
```

Calling the same procedure in FBD:

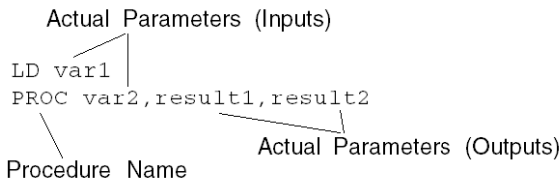


Informal Call without CAL Instruction

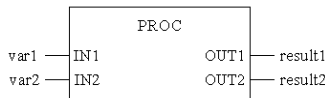
With this type of call (call without formal parameter names), procedures are called using an instruction sequence consisting of the first actual parameter loaded into the accumulator, followed by the procedure name, followed by a list of the input and output actual parameters. The order in which the actual parameters are listed is **significant**. The list of actual parameters cannot be wrapped.

EN and ENO **cannot** be used for this type of call.

Calling a procedure with formal parameter names:



Calling the same procedure in FBD:



NOTE: Note that when making an informal call, the list of actual parameters **cannot** be put in brackets. IEC 61133-3 requires that the brackets be left out in this case to illustrate that the first actual parameter is not a part of the list.

Invalid informal call for a procedure:

```
LD A
LIMIT (B,C)
```

If the value to be processed (first actual parameter) is already in the accumulator, the load instruction can be omitted.

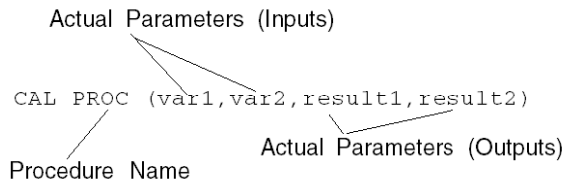
```
EXAMP1 var2, result1, result2
```

Informal Call with CAL Instruction

With this type of call, procedures are called using an instruction sequence consisting of the CAL instruction, followed by the procedure name followed by a list of the input and output actual parameters. The order in which the actual parameters are listed is **significant**. The list of actual parameters cannot be wrapped.

EN and ENO **cannot** be used for this type of call.

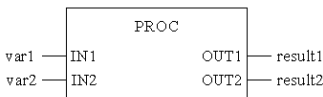
Calling a procedure with formal parameter names using CAL instruction:



or

```
CAL PROC (var1,
          var2,
          result1,
          result2)
```

Calling the same procedure in FBD:



NOTE: Unlike informal calls without a CAL instruction, when making informal calls with a CAL instruction, the value to be processed (first actual parameter) is not explicitly loaded in the battery. Instead it is part of the list of actual parameters. For this reason, when making informal calls with a CAL instruction, the list of actual parameters must be put in brackets.

EN and ENO

With all procedures, an EN input and an ENO output can be configured.

If the value of EN is equal to "0" when the procedure is called, the algorithms defined by the procedure are not executed and ENO is set to "0".

If the value of EN is "1" when the procedure is called, the algorithms defined by the function are executed. After the algorithms have been executed successfully, the value of ENO is set to "1". If an error occurs when executing these algorithms, ENO is set to "0".

If the EN pin is not assigned a value, when the FFB is invoked, the algorithm defined by the FFB is executed (same as if EN equals to "1").

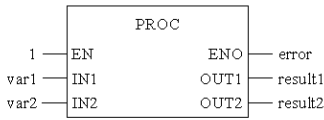
If ENO is set to "0" (caused when EN=0 or an error occurred during executing), the outputs of the procedure are set to "0".

If EN/ENO are used, the procedure call must be formal. The assignment of variables to ENO must be made using the => operator.

```
PROC (EN:=1, IN1:=var1, IN2:=var2,
```

```
ENO=>error, OUT1=>result1, OUT2=>result2) ;
```

Calling the same procedure in FBD:



VAR_IN_OUT Variable

Procedures are often used to read a variable at an input (input variables), to process it and to output the updated values of the same variable (output variables). This special type of input/output variable is also called a VAR_IN_OUT variable.

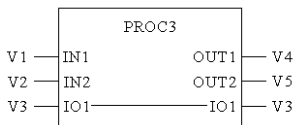
The following special features are to be noted when using procedures with VAR_IN_OUT variables.

- All VAR_IN_OUT inputs must be assigned a variable.
- VAR_IN_OUT inputs may not have literals or constants assigned to them.
- VAR_IN_OUT outputs may **not** have values assigned to them.
- VAR_IN_OUT variables **cannot** be used outside of the procedure call.

Calling a procedure with VAR_IN_OUT variable in IL:

```
PROC3 (IN1:=V1, IN2:=V2, IO1:=V3,
      OUT1=>V4, OUT2=>V5) ;
```

Calling the same procedure in FBD:



VAR_IN_OUT variables **cannot** be used outside the procedure call.

The following procedure calls are therefore **invalid**:

Invalid call, example 1:

| | |
|---------------|---|
| LD V1 | Loading a V1 variable in the accumulator |
| CAL InOutProc | Calling a procedure with the VAR_IN_OUT parameter. The accumulator now contains a reference to a VAR_IN_OUT parameter. |
| AND V2 | AND operation on contents of accumulator with variable V2. Error: The operation cannot be carried out since the VAR_IN_OUT parameter (contents of accumulator) cannot be accessed outside the procedure call. |

Invalid call, example 2:

| | |
|---------------------|---|
| LD V1 | Loading a V1 variable in the accumulator |
| AND InOutProc.inout | AND operation on the contents of the accumulator and a reference to a VAR_IN_OUT parameter. Fehler: The operation cannot be carried out since the VAR_IN_OUT parameter cannot be accessed outside the procedure call. |

Invalid call, example 3:

| | |
|------------|--|
| LD V1 | Loading a V1 variable in the accumulator |
| InOutFB V2 | Calling the procedure with assignment of the actual parameter (V2) to the VAR_IN_OUT parameter. Error: The operation cannot be carried out as with this type of procedure call only the VAR_IN_OUT parameter would be stored in the accumulator for later use. |

The following procedure calls are always **valid**:

Valid call, example 1:

| | |
|------------------------------------|--|
| CAL InOutProc (IN1:=V1, inout:=V2) | Calling a procedure with the VAR_IN_OUT parameter and formal assignment of the actual parameter within the procedure call. |
|------------------------------------|--|

Valid call, example 2:

| | |
|--------------------------------|--|
| InOutProc (IN1:=V1, inout:=V2) | Calling a procedure with the VAR_IN_OUT parameter and formal assignment of the actual parameter within the procedure call. |
|--------------------------------|--|

Valid call, example 3:

| | |
|------------------------|--|
| CAL InOutProc (V1, V2) | Calling a procedure with the VAR_IN_OUT parameter and informal assignment of the actual parameter within the procedure call. |
|------------------------|--|

Structured Text (ST)

What's in This Chapter

| | |
|--|-----|
| General Information about the Structured Text ST | 420 |
| Instructions..... | 429 |
| Calling Elementary Functions, Elementary Function Blocks, Derived Function Blocks and Procedures..... | 442 |

Overview

This chapter describes the programming language structured text ST which conforms to IEC 61131.

General Information about the Structured Text ST

Overview

This section contains a general overview of the structured text ST.

General Information about Structured Text (ST)

Introduction

With the programming language of structured text (ST), it is possible, for example, to call up function blocks, perform functions and assignments, conditionally perform instructions and repeat tasks.

Expression

The ST programming language works with "Expressions".

Expressions are constructions consisting of operators and operands that return a value when executed.

Operator

Operators are symbols representing the operations to be executed.

Operand

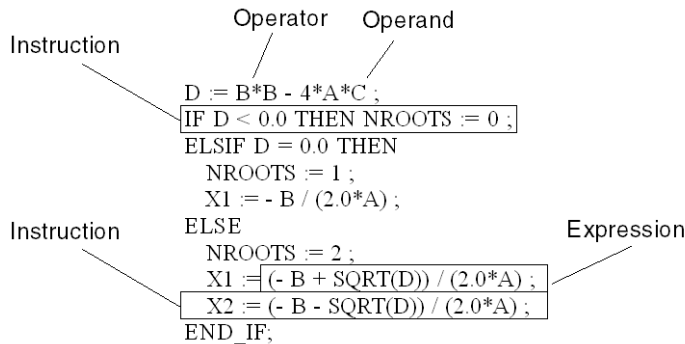
Operators are used for operands. Operands are variables, literals, FFB inputs/outputs etc.

Instructions

Instructions are used to assign the values returned from the expressions to actual parameters and to structure and control the expressions.

Representation of an ST Section

Representation of an ST section:



Section Size

The length of an instruction line is limited to 300 characters.

The length of an ST section is not limited within the programming environment. The length of an ST section is usually limited by the size of the PLC memory.

NOTE: There is no size limitation for section, but sometime when using a large amount of literal assignments or some specific instructions, a section can generate a Code generation failure during an application build. Then the solution is to split the section in two or more sections to build the application.

Syntax

Identifiers and Keywords are not case sensitive.

Exception: Not allowed - spaces and tabs

- keywords

- literals
- values
- identifiers
- variables and
- limiter combinations [e.g. (* for comments)]

Execution Sequence

The evaluation of an expression consists of applying the operators to the operands in the sequence as defined by the rank of the operators (see [Table of Operators, page 425](#)). The operator with the highest rank in an expression is performed first, followed by the operator with the next highest rank, etc., until the evaluation is complete. Operators with the same rank are performed from left to right, as they are written in the expression. This sequence can be altered with the use of parentheses.

If, for example, A, B, C and D have the values 1, 2, 3 and 4, and are calculated as follows:
 $A+B-C*D$

the result is -9.

In the case of the following calculation:

$(A+B-C) * D$

the result is 0.

If an operator contains two operands, the left operand is executed first, e.g. in the expression

$SIN(A) * COS(B)$

the expression $SIN(A)$ is calculated first, then $COS(B)$ and only then is the product calculated.

Error Behavior

The following conditions are handled as an error when executing an expression:

- Attempting to divide by 0.
- Operands do not contain the correct data type for the operation.
- The result of a numerical operation exceeds the value range of its data type

If an error occurs when executing the operation, the corresponding Systembit (%S) is set (if supported by the PLC being used).

IEC Conformity

For a description of IEC conformity for the ST programming language, see IEC Conformity, page 508.

Operands

Introduction

An operand can be:

- an address
- a literal
- a variable
- a multi-element variable
- an element of a multi-element variable
- a function call
- an FFB output

Data Types

Data types, which are in an instruction of processing operands, must be identical. Should operands of various types be processed, a type conversion must be performed beforehand.

In the example the integer variable `i1` is converted into a real variable before being added to the real variable `r4`.

```
r3 := r4 + SIN(INT_TO_REAL(i1)) ;
```

As an exception to this rule, variables with data type `TIME` can be multiplied or divided by variables with data type `INT`, `DINT`, `UINT` or `UDINT`.

Permitted operations:

- `timeVar1 := timeVar2 / dintVar1;`
- `timeVar1 := timeVar2 * intVar1;`
- `timeVar := 10 * time#10s;`

This function is listed by IEC 61131-3 as "undesired" service.

Direct Use of Addresses

Addresses can be used directly (without a previous declaration). In this case the addresses data type is assigned directly. The assignment is made using the "Large prefix".

The different large prefixes are given in the following table:

| Large prefix / Symbol | Example | Data type |
|-----------------------|----------------------------------|-----------|
| no prefix | %I10, %CH203.MOD, %CH203.MOD.ERR | BOOL |
| X | %MX20 | BOOL |
| B | %QB102.3 | BYTE |
| W | %KW43 | INT |
| D | %QD100 | DINT |
| F | %MF100 | REAL |

Using Other Data Types

Should other data types be assigned as the default data types of an address, this must be done through an explicit declaration. This variable declaration takes place comfortably using the variable editor. The data type of an address can not be declared directly in an ST section (e.g. declaration AT %MW1: UINT; not permitted).

For example, the following variables are declared in the variable editor:

```
UnlocV1: ARRAY [1..10] OF INT;
LocV1:   ARRAY [1..10] OF INT AT %MW100;
LocV2:   TIME AT %MW100;
```

The following calls then have the correct syntax:

```
%MW200 := 5;
UnlocV1[2] := LocV1[%MW200];
LocV2      := t#3s;
```

Accessing Field Variables

When accessing field variables (ARRAY), only literals and variables of the INT, UINT, DINT and UDINT data types are permitted in the index entry.

The index of an ARRAY element can be negative if the lower threshold of the range is negative.

Example: Using field variables

```
var1[i] := 8 ;
var2.otto[4] := var3 ;
var4[1+i+j*5] := 4 ;
```


Operators

Introduction

An operator is a symbol for:

- an arithmetic operation to be executed or
- a logical operation to be executed or
- a function edit (call)

Operators are generic, i.e. they adapt automatically to the data type of the operands.

Table of Operators

Operators are executed in sequence according to priority, see also [Execution Sequence](#), page 422.

ST programming language operators:

| Operator | Meaning | Order of rank | possible operands | Description |
|------------------------------------|----------------------------|---------------|---|---|
| () | Use of Brackets: | 1 (highest) | Expression | Brackets are used to alter the execution sequence of the operators. Example: If the operands A, B, C and D have the values 1, 2, 3, and 4, A+B-C*D has the result -9 and (A+B-C) *D has the result 0. |
| FUNC-NAME (Actual parameter -list) | Function processing (call) | 2 | Expression, Literal, Variable, Address (all data types) | Function processing is used to execute functions (see Calling Elementary Functions , page 442). |
| - | Negation | 3 | Expression, Literal, Variable, Address of Data Type INT, DINT or REAL | During negation - a sign reversal for the value of the operand takes place. Example: In the example OUT is -4 if IN1 is 4. OUT := - IN1 ; |
| NOT | Complement | 3 | Expression, Literal, Variable, Address of Data Type BOOL, BYTE, WORD or DWORD | In NOT a bit by bit inversion of the operands takes place. Example: In the example OUT is 0011001100 if IN1 is 1100110011. |

| Operator | Meaning | Order of rank | possible operands | Description |
|----------|----------------|---------------|---|---|
| | | | | <p>OUT := NOT IN1 ;</p> |
| ** | Exponentiation | 4 | Expression, Literal, Variable, Address of Data Type REAL (Basis) and INT, DINT, UINT, UDINT or REAL (Exponent) | <p>In exponentiation, ** the value of the first operand (basis) is raised to the power of the second operand (exponent).</p> <p>Example: In the example OUT is 625.0 if IN1 is 5.0 and IN2 is 4.0.</p> <p>OUT := IN1 ** IN2 ;</p> |
| * | Multiplication | 5 | Expression, Literal, Variable, Address of Data Type INT, DINT, UINT, UDINT or REAL | <p>In multiplication, * the value of the first operand is multiplied by the value of the second operand (exponent) .</p> <p>Example: In the example OUT is 20.0 if IN1 is 5.0 and IN2 is 4.0.</p> <p>OUT := IN1 * IN2 ;</p> <p>Note: The MULTIME function in the obsolete library is available for multiplications involving the data type Time.</p> |
| / | Division | 5 | Expression, Literal, Variable, Address of Data Type INT, DINT, UINT, UDINT or REAL | <p>In division, / the value of the first operand is divided by the value of the second operand.</p> <p>Example: In the example OUT is 4.0 if IN1 is 20.0 and IN2 is 5.0.</p> <p>OUT := IN1 / IN2 ;</p> <p>Note: The DIVTIME function in the obsolete library is available for divisions involving the data type Time.</p> |
| MOD | Modulo | 5 | Expression, Literal, Variable, Address of Data Type INT, DINT, UINT or UDINT | <p>For MOD the value of the first operand is divided by that of the second operand and the remainder of the division (Modulo) is displayed as the result.</p> <p>Example: In this example</p> <ul style="list-style-type: none"> • OUT is 1 if IN1 is 7 and IN2 is 2 • OUT is 1 if IN1 is 7 and IN2 is -2 • OUT is -1 if IN1 is -7 and IN2 is 2 • OUT is -1 if IN1 is -7 and IN2 is -2 <p>OUT := IN1 MOD IN2 ;</p> |
| + | Addition | 6 | Expression, Literal, Variable, Address of Data Type INT, DINT, UINT, UDINT, REAL or TIME | <p>In addition, + the value of the first operand is added to the value of the second operand.</p> <p>Example: In this example</p> <p>OUT is 9, if IN1 is 7 and IN2 is 2</p> |

| Operator | Meaning | Order of rank | possible operands | Description |
|----------|-------------------------------------|---------------|---|---|
| | | | | OUT := IN1 + IN2 ; |
| - | Subtraction | 6 | Expression, Literal, Variable, Address of Data Type INT, DINT, UINT, UDINT, REAL or TIME | In subtraction, - the value of the second operand is subtracted from the value of the first operand. Example: In the example OUT is 6 if IN1 is 10 and IN2 is 4. OUT := IN1 - IN2 ; |
| < | Less than comparison | 7 | Expression, Literal, Variable, Address of Data Type BOOL, BYTE, INT, DINT, UINT, UDINT, REAL, TIME, WORD, DWORD, STRING, DT, DATE or TOD | The value of the first operand is compared with the value of the second using <. If the value of the first operand is less than the value of the second, the result is a Boolean 1. If the value of the first operand is greater than or equal to the value of the second, the result is a Boolean 0. Example: In the example OUT is 1 if IN1 is less than 10 and is otherwise 0. OUT := IN1 < 10 ; |
| > | Greater than comparison | 7 | Expression, Literal, Variable, Address of Data Type BOOL, BYTE, INT, DINT, UINT, UDINT, REAL, TIME, WORD, DWORD, STRING, DT, DATE or TOD | The value of the first operand is compared with the value of the second using >. If the value of the first operand is greater than the value of the second, the result is a Boolean 1. If the value of the first operand is less than or equal to the value of the second, the result is a Boolean 0. Example: In the example OUT is 1 if IN1 is greater than 10, and is 0 if IN1 is less than 0. OUT := IN1 > 10 ; |
| <= | Less than or equal to comparison | 7 | Expression, Literal, Variable, Address of Data Type BOOL, BYTE, INT, DINT, UINT, UDINT, REAL, TIME, WORD, DWORD, STRING, DT, DATE or TOD | The value of the first operand is compared with the value of the second operand using <=. If the value of the first operand is less than or equal to the value of the second, the result is a Boolean 1. If the value of the first operand is greater than the value of the second, the result is a Boolean 0. Example: In the example OUT is 1 if IN1 is less than or equal to 10, and otherwise is 0. OUT := IN1 <= 10 ; |
| >= | Greater than or equal to comparison | 7 | Expression, Literal, Variable, Address of Data Type BOOL, BYTE, INT, DINT, UINT, UDINT, REAL, TIME, WORD, DWORD, STRING, DT, DATE or TOD | The value of the first operand is compared with the value of the second operand using >=. If the value of the first operand is greater than or equal to the value of the second, the result is a Boolean 1. If the value of the first |

| Operator | Meaning | Order of rank | possible operands | Description |
|----------|-------------|---------------|---|--|
| | | | | operand is less than the value of the second, the result is a Boolean 0. Example: In the example OUT is 1 if IN1 is greater than or equal to 10, and otherwise is 0. OUT := IN1 >= 10 ; |
| = | Equality | 8 | Expression, Literal, Variable, Address of Data Type BOOL, BYTE, INT, DINT, UINT, UDINT, REAL, TIME, WORD, DWORD, STRING, DT, DATE or TOD | The value of the first operand is compared with the value of the second operand using =. If the value of the first operand is equal to the value of the second, the result is a Boolean 1. If the value of the first operand is not equal to the value of the second, the result is a Boolean 0. Example: In the example OUT is 1 if IN1 is equal to 10 and is otherwise 0. OUT := IN1 = 10 ; |
| <> | Inequality | 8 | Expression, Literal, Variable, Address of Data Type BOOL, BYTE, INT, DINT, UINT, UDINT, REAL, TIME, WORD, DWORD, STRING, DT, DATE or TOD | The value of the first operand is compared with the value of the second using <>. If the value of the first operand is not equal to the value of the second, the result is a Boolean 1. If the value of the first operand is equal to the value of the second, the result is a Boolean 0. Example: In the example OUT is 1 if IN1 is not equal to 10 and is otherwise 0. OUT := IN1 <> 10 ; |
| & | Logical AND | 9 | Expression, Literal, Variable, Address of Data Type BOOL, BYTE, WORD or DWORD | With &, there is a logical AND link between the operands. In the case of BYTE, WORD and DWORD data types, the link is made bit by bit. Example: In the examples OUT is 1 if IN1, IN2 and IN3 are 1. OUT := IN1 & IN2 & IN3 ; |
| AND | Logical AND | 9 | Expression, Literal, Variable, Address of Data Type BOOL, BYTE, WORD or DWORD | With AND, there is a logical AND link between the operands. In the case of BYTE, WORD and DWORD data types, the link is made bit by bit. Example: In the examples OUT is 1 if IN1, IN2 and IN3 are 1. OUT := IN1 AND IN2 AND IN3 ; |

| Operator | Meaning | Order of rank | possible operands | Description |
|----------|----------------------|---------------|--|---|
| XOR | Logical Exclusive OR | 10 | Expression, Literal, Variable, Address of Data Type BOOL, BYTE, WORD or DWORD | <p>With XOR, there is a logical Exclusive OR link between the operations. In the case of BYTE, WORD and DWORD data types, the link is made bit by bit.</p> <p>Example: In the example OUT is 1 if IN1 and IN2 are not equal. If A and B have the same status (both 0 or 1), D is 0.</p> <p>OUT := IN1 XOR IN2 ;</p> <p>If more than two operands are linked, the result with an uneven number of 1-states is 1, and is 0 with an even number of 1-states.</p> <p>Example: In the example OUT is 1 if 1 or 3 operands are 1. OUT is 0 if 0, 2 or 4 operands are 1.</p> <p>OUT := IN1 XOR IN2 XOR IN3 XOR IN4 ;</p> |
| OR | Logical OR | 11 (lowest) | Expression, Literal, Variable, Address of Data Type BOOL, BYTE, WORD or DWORD | <p>With OR, there is a logical OR link between the operands. With the BYTE and WORD, DWORD data types, the link is made bit by bit.</p> <p>Example: In the example OUT is 1 if IN1, IN2 or IN3 is 1.</p> <p>OUT := IN1 OR IN2 OR IN3 ;</p> |

Instructions

Overview

This section describes the instructions for the programming language of structured text ST.

Instructions

Description

Instructions are the "Commands" of the ST programming language.

Instructions must be terminated with semicolons.

Several instructions (separated by semicolons) can be present in one line.

A single semicolon represents an Empty instruction, page 441.

Assignment

Introduction

When an assignment is performed, the current value of a single or multi-element variable is replaced by the result of the evaluation of the expression.

An assignment consists of a variable specification on the left side, followed by the assignment operator `:=`, followed by the expression to be evaluated.

Both variables (left and right sides of the assignment operator) must have the same data type.

Arrays are a special case. After being explicitly enabled, assignment of two arrays with different lengths can be made.

Assigning the Value of a Variable to Another Variable

Assignments are used to assign the value of a variable to another variable.

The instruction

```
A := B ;
```

is used, for example, to replace the value of the variable `A` with the current value of variable `B`. If `A` and `B` are elementary data types, the individual value of `B` is passed to `A`. If `A` and `B` are derived data types, the values of all `B` elements are passed to `A`.

Assigning the Value of a Literal to a Variable

Assignments are used to assign a literal to variables.

The instruction

```
C := 25 ;
```

is used, for example, to assign the value 25 to the variable `C`.

Assigning the Value of an Operation to a Variable

Assignments are used to assign to a variable a value which is the result of an operation.

The instruction

```
X := (A+B-C) *D ;
```

is used, for example, to assign the result of the operation $(A+B-C) *D$ to the variable `X`.

Assigning the Value of an FFB to a Variable

Assignments are used to assign a value returned by a function or a function block to a variable.

The instruction

```
B := MOD(C, A) ;
```

is used, for example, to call the MOD (Modulo) function and assign the result of the calculation to the variable B.

The instruction

```
A := MY_TON.Q ;
```

is used, for example, to assign the value of the Q output of the MY_TON function block (instance of the TON function block) to the variable A. (This is not a function block call)

Multiple Assignments

Multiple assignments are a supplement to IEC 61131-3 and must be enabled explicitly.

Even after being enabled, multiple assignments are NOT allowed in the following cases:

- in the parameter list for a function block call
- in the element list to initialize structured variables

The instruction

```
X := Y := Z
```

is allowed.

The instructions

```
FB(in1 := 1, In2 := In3 := 2) ;
```

and

```
strucVar := (comp1 := 1, comp2 := comp3 := 2) ;
```

are not allowed.

Assignments between Arrays and WORD-/DWORD Variables

Assignments between arrays and WORD-/DWORD variables are only possible if a type conversion has previously been carried out, e.g.:

```
%Q3.0:16 := INT_TO_AR_BOOL(%MW20) ;
```

The following conversion functions are available (General Library, family Array):

- MOVE_BOOL_AREBOOL
- MOVE_WORD_ARWORD

- MOVE_DWORD_ARDWORD
- MOVE_INT_ARINT
- MOVE_DINT_ARDINT
- MOVE_REAL_ARREAL

Select Instruction IF...THEN...END_IF

Description

The IF instruction determines that an instruction or a group of instructions will only be executed if its related Boolean expression has the value 1 (true). If the condition is 0 (false), the instruction or the instruction group will not be executed.

The THEN instruction identifies the end of the condition and the beginning of the instruction(s).

The END_IF instruction marks the end of the instruction(s).

NOTE: 74 IF...THEN...END_IF instructions may be nested to generate complex selection instructions.

Example IF...THEN...END_IF

The condition can be expressed using a Boolean variable.

If FLAG is 1, the instructions will be executed; if FLAG is 0, they will not be executed.

```
IF FLAG THEN
  C:=SIN(A) * COS(B) ;
  B:=C - A ;
END_IF ;
```

The condition can be expressed using an operation that returns a Boolean result.

If A is greater than B, the instructions will be executed; if A is less than or equal to B, they will not be executed.

```
IF A>B THEN
  C:=SIN(A) * COS(B) ;
  B:=C - A ;
END_IF ;
```

Example IF NOT...THEN...END_IF

The condition can be inverted using NOT (execution of both instructions at 0).

```
IF NOT FLAG THEN
  C:=SIN_REAL(A) * COS_REAL(B) ;
```



```
    B:=C - A ;  
END_IF ;
```

See Also

ELSE, page 433

ELSIF, page 434

Select Instruction ELSE

Description

The ELSE instruction always comes after an IF . . . THEN, ELSIF . . . THEN or CASE instruction.

If the ELSE instruction comes after an IF or ELSIF instruction, the instruction or group of instructions will only be executed if the associated Boolean expressions of the IF and ELSIF instruction are 0 (false). If the condition of the IF or ELSIF instruction is 1 (true), the instruction or group of instructions will not be executed.

If the ELSE instruction comes after CASE, the instruction or group of instructions will only be executed if no tag contains the value of the selector. If an identification contains the value of the selector, the instruction or group of instructions will not be executed.

NOTE: Any number of IF . . . THEN . . . ELSE . . . END_IF instructions may be nested to generate complex selection instructions.

Example ELSE

```
IF A>B THEN  
    C:=SIN(A) * COS(B) ;  
    B:=C - A ;  
ELSE  
    C:=A + B ;  
    B:=C * A ;  
END_IF ;
```

See Also

IF, page 432

ELSIF, page 434

CASE, page 435

Select Instruction ELSIF...THEN

Description

The `ELSE` instruction always comes after an `IF...THEN` instruction. The `ELSIF` instruction determines that an instruction or group of instructions is only executed if the associated Boolean expression for the `IF` instruction has the value 0 (false) and the associated Boolean expression of the `ELSIF` instruction has the value 1 (true). If the condition of the `IF` instruction is 1 (true) or the condition of the `ELSIF` instruction is 0 (false), the command or group of commands will not be executed.

The `THEN` instruction identifies the end of the `ELSIF` condition(s) and the beginning of the instruction(s).

NOTE: Any number of `IF...THEN...ELSIF...THEN...END_IF` instructions may be nested to generate complex selection instructions.

Example ELSIF...THEN

```
IF A>B THEN
    C:=SIN(A) * COS(B) ;
    B:=SUB(C,A) ;
ELSIF A=B THEN
    C:=ADD(A,B) ;
    B:=MUL(C,A) ;
END_IF ;
```

For Example Nested Instructions

```
IF A>B THEN
    IF B=C THEN
        C:=SIN(A) * COS(B) ;
    ELSE
        B:=SUB(C,A) ;
    END_IF ;
ELSIF A=B THEN
    C:=ADD(A,B) ;
    B:=MUL(C,A) ;
ELSE
    C:=DIV(A,B) ;
END_IF ;
```

See Also

[IF](#), page 432

ELSE, page 433

Select Instruction CASE...OF...END_CASE

Description

The `CASE` instruction consists of an `INT` data type expression (the "selector") and a list of instruction groups. Each group is provided with a tag which consists of one or several whole numbers (`INT`, `DINT`, `UINT`, `UDINT`) or ranges of whole number values. The first group is executed by instructions, whose tag contains the calculated value of the selector. Otherwise none of the instructions will be executed.

The `OF` instruction indicates the start of the tag.

An `ELSE` instruction may be carried out within the `CASE` instruction, whose instructions are executed if no tag contains the selector value.

The `END_CASE` instruction marks the end of the instruction(s).

Example CASE . . . OF . . . END_CASE

Example `CASE . . . OF . . . END_CASE`

```

Selector
  |
  v
CASE SELECT OF
1, 5:  C:=SIN(A) * COS(B) ;
2:     B:=C - A ;
6..10: C:=C * A ;
ELSE
  B:=C * A ;
  C:=A / B ;
END_CASE ;
  
```

See Also

ELSE, page 433

Repeat Instruction FOR...TO...BY...DO...END_FOR

Description

The `FOR` instruction is used when the number of occurrences can be determined in advance. Otherwise `WHILE`, page 438 or `REPEAT`, page 438 are used.

The `FOR` instruction repeats an instruction sequence until the `END_FOR` instruction. The number of occurrences is determined by start value, end value and control variable.

The control variable, initial value and end value must be of the same data type (`INT`, `UINT`, `DINT` or `UDINT`).

The control variable, initial value and end value can be changed by a repeated instruction. This is a supplement to IEC 61131-3.

The `FOR` instruction increments the control variable value of one start value to an end value. The increment value has the default value 1. If a different value is to be used, it is possible to specify an explicit increment value (variable or constant). The control variable value is checked before each new loop. If it is outside the start value and end value range, the loop will be left.

Before running the loop for the first time a check is made to determine whether incrementation of the control variables, starting from the initial value, is moving toward the end value. If this is not the case (e.g. initial value \leq end value and negative increment), the loop will not be processed. The control variable value is not defined outside of the loop.

The `DO` instruction identifies the end of the repeat definition and the beginning of the instruction(s).

The occurrence may be terminated early using the `EXIT`. The `END_FOR` instruction marks the end of the instruction(s).

Example: FOR with Increment 1

FOR with increment 1

```

FOR i := 1 TO 50 DO
  C := C * COS(B) ;
END_FOR ;
    
```

FOR with Increment not Equal to 1

If an increment other than 1 is to be used, it can be defined by `BY`. The increment, the initial value, the end value and the control variable must be of the same data type (`DINT` or `INT`). The criterion for the processing direction (forwards, backwards) is the sign of the `BY` expression. If this expression is positive, the loop will run forward; if it is negative, the loop will run backward.

Example: Counting forward in Two Steps

Counting forward in two steps

Control variable Start value End value Increment

```

FOR i:= 1 TO 10 BY 2 DO (* BY > 0 : Forwards.loop *)
  C:= C * COS(B) ; (* Loop is 5 x executed *)
END_FOR ;

```

Example: Counting Backwards

Counting backwards

```

FOR i:= 10 TO 1 BY -1 DO (* BY < 0 : Backwards.loop *)
  C:= C * COS(B) ; (* Instruction is executed 10 x *)
END_FOR ;

```

Example: "Unique" Loops

The loops in the example are run exactly once, as the initial value = end value. In this context it does not matter whether the increment is positive or negative.

```

FOR i:= 10 TO 10 DO (* Unique Loop *)
  C:= C * COS(B) ;
END_FOR ;

OR

FOR i:= 10 TO 10 BY -1 DO (* Unique Loop *)
  C:= C * COS(B) ;
END_FOR ;

```

Example: Critical Loops

If the increment is $j > 0$ in the example, the instruction is executed.

If $j < 0$, the instructions are not executed because the situation initial value $<$ only allows the end value to be incremented by ≥ 0 .

If $j = 0$, the instructions are executed and an endless loop is created as the end value will never be reached with an increment of 0.

```

FOR i:= 1 TO 10 BY j DO
  C:= C * COS(B) ;
END_FOR ;

```

Repeat Instruction WHILE...DO...END_WHILE

Description

The `WHILE` instruction has the effect that a sequence of instructions will be executed repeatedly until its related Boolean expression is 0 (false). If the expression is false right from the start, the group of instructions will not be executed at all.

The `DO` instruction identifies the end of the repeat definition and the beginning of the instruction(s).

The occurrence may be terminated early using the `EXIT`.

The `END_WHILE` instruction marks the end of the instruction(s).

In the following cases `WHILE` may not be used as it can create an endless loop which causes the program to crash:

- `WHILE` may not be used for synchronization between processes, e.g. as a "Waiting Loop" with an externally defined end condition.
- `WHILE` may not be used in an algorithm, as the completion of the loop end condition or execution of an `EXIT` instruction can not be guaranteed.

Example WHILE...DO...END_WHILE

```
x := 1;
WHILE x <= 100 DO
    x := x + 4;
END_WHILE ;
```

See Also

`EXIT`, page 439

Repeat Instruction REPEAT...UNTIL...END_REPEAT

Description

The `REPEAT` instruction has the effect that a sequence of instructions is executed repeatedly (at least once), until its related Boolean condition is 1 (true).

The `UNTIL` instruction marks the end condition.

The occurrence may be terminated early using the `EXIT`.

The `END_REPEAT` instruction marks the end of the instruction(s).

In the following cases `REPEAT` may not be used as it can create an endless loop which causes the program to crash:

- `REPEAT` may not be used for synchronization between processes, e.g., as a "Waiting Loop" with an externally defined end condition.
- `REPEAT` may not be used in an algorithm, such as the completion of the loop end condition or execution of an `EXIT` instruction cannot be guaranteed.

Example `REPEAT . . . UNTIL . . . END_REPEAT`

```
x := -1;
REPEAT x := x + 2;
UNTIL x >= 101
END_REPEAT;
```

See Also

`EXIT`, page 439

Repeat Instruction `EXIT`

Description

The `EXIT` instruction is used to terminate repeat instructions (`FOR`, `WHILE`, `REPEAT`) before the end condition has been met.

If the `EXIT` instruction is within a nested repetition, the innermost loop (in which `EXIT` is situated) is left. Next, the first instruction following the loop end (`END_FOR`, `END_WHILE` or `END_REPEAT`) is executed.

Example `EXIT`

If `FLAG` has the value 0, `SUM` will be 15 following the execution of the instructions.

If `FLAG` has the value 1, `SUM` will be 6 following the execution of the instructions.

```
SUM := 0 ;
FOR I := 1 TO 3 DO
  FOR J := 1 TO 2 DO
    IF FLAG=1 THEN EXIT ;
  END_IF ;
  SUM := SUM + J ;
END_FOR ;
```

```
SUM := SUM + I ;  
END_FOR ;
```

See Also

CASE, page 435

WHILE, page 438

REPEAT, page 438

Subroutine Call

Subroutine Call

A subroutine call consists of the name of the subroutine section followed by an empty parameter list.

Subroutine calls do not return a value.

The subroutine to be called must be located in the same task as the ST section called.

Subroutines can also be called from within subroutines.

For example:

```
SubroutineName () ;
```

Subroutine calls are a supplement to IEC 61131-3 and must be enabled explicitly.

In SFC action sections, subroutine calls are only allowed when Multitoken Operation is enabled.

RETURN

Description

RETURN instructions can be used in DFBs (derived function blocks) and in SRs (subroutines).

RETURN instructions can not be used in the main program.

- In a DFB, a RETURN instruction forces the return to the program which called the DFB.
 - The rest of the DFB section containing the RETURN instruction is not executed.
 - The next sections of the DFB are not executed.

The program which called the DFB will be executed after return from the DFB.

If the DFB is called by another DFB, the calling DFB will be executed after return.

- In a SR, a `RETURN` instruction forces the return to the program which called the SR.
 - The rest of the SR containing the `RETURN` instruction is not executed.The program which called the SR will be executed after return from the SR.

Empty Instruction

Description

A single semicolon `;` represents an empty instruction.

For example,

```
IF x THEN ; ELSE ..
```

In this example, an empty instruction follows the `THEN` instruction. This means that the program exits the `IF` instruction as soon as the `IF` condition is 1.

Labels and Jumps

Introduction

Labels serve as destinations for jumps.

Jumps and labels in ST are a supplement to the IEC 61131-3 and must be enabled explicitly.

Label Properties

Label properties:

- Labels must always be the first element in a line.
- Labels may only come before instructions of the first order (not in loops).
- The name must be clear throughout the directory, and it is not upper/lower case sensitive.
- Labels must conform to the general naming conventions.
- Labels are separated by a colon `:` from the following instruction.

Properties of Jumps

Properties of jumps

- Jumps can be made within program and DFB sections.
- Jumps are only possible in the current section.

Example

```
IF var1 THEN
  JMP START;
:
:START: ...
```

Comment

Description

In the ST editor, comments always start with the string `(* and end in the string *) . Any comments can be entered between these character strings. Comments can be entered in any position in the ST editor, except in keywords, literals, identifiers and variables.`

Nesting comments is not permitted according to IEC 61131-3. If comments are nested nevertheless, then they must be enabled explicitly.

Calling Elementary Functions, Elementary Function Blocks, Derived Function Blocks and Procedures

Overview

Calling Elementary Functions, Elementary Function Blocks, Derived Function Blocks and Procedures in the ST programming language.

Calling Elementary Functions

Elementary Functions

Elementary functions are provided in the form of libraries. The logic of the functions is created in the programming language C and may not be modified in the ST editor.

Functions have no internal states. If the input values are the same, the value at the output is the same for all executions of the function. For example, the addition of two values gives the same result at every execution.

Some elementary functions can be extended to more than 2 inputs.

Elementary functions only have one return value (Output).

Parameters

"Inputs" and one "output" are required to transfer values to or from a function. These are called formal parameters.

The current process states are transferred to the formal parameters. These are called actual parameters.

The following can be used as actual parameters for function inputs:

- Variable
- Address
- Literal
- ST Expression

The following can be used as actual parameters for function outputs:

- Variable
- Address

The data type of the actual parameters must match the data type of the formal parameters. The only exceptions are generic formal parameters whose data type is determined by the actual parameter.

When dealing with generic `ANY_BIT` formal parameters, actual parameters of the `INT` or `DINT` (not `UINT` and `UDINT`) data types can be used.

This is a supplement to IEC 61131-3 and must be enabled explicitly.

Example:

Allowed:

```
AND (AnyBitParam := IntVar1, AnyBitParam2 := IntVar2);
```

Not allowed:

```
AND_WORD (WordParam1 := IntVar1, WordParam2 := IntVar2);
```

(In this case, `AND_INT` must be used.)

```
AND_ARRAY_WORD (ArrayInt, ...);
```

(In this case an explicit type conversion must be carried out using `INT_ARR_TO_WORD_ARR (...)`).

Not all formal parameters must be assigned with a value. The formal parameter types that must be assigned with a value are in this table:

| Parameter type | EDT | STRING | ARRAY | ANY_ARRAY | IODDT | STRUC-T | FB | ANY |
|--|-----|--------|-------|-----------|-------|---------|----|-----|
| Input | - | - | - | - | + | - | + | - |
| VAR_IN_OUT | + | + | + | + | + | + | / | + |
| Output | - | - | - | - | - | - | / | - |
| + Actual parameter required | | | | | | | | |
| - Actual parameter not required, it's the general rule, but there are exceptions for some FFBs, for instance when some parameters are used to characterize the information we want to be given by the FFB. | | | | | | | | |
| / not applicable | | | | | | | | |

If no value is allocated to a formal parameter, then the initial value is used for executing the function block. If no initial value has been defined, then the default value (0) is used.

Programming Notes

Attention should be paid to the following:

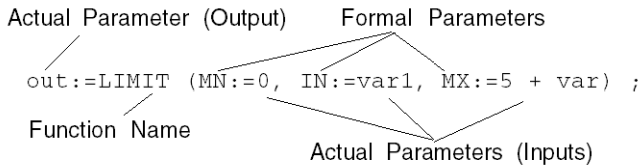
- All generic functions are overloaded. This means the functions can be called with or without entering the data type.
E.g.
`i1 := ADD (i2, 3);`
is identical to
`i1 := ADD_INT (i2, 3);`
- Functions can be nested, page 446.
- Functions are only executed if the input `EN = 1` or the `EN`, page 447 input is not used.
- There are two ways of calling a function:
 - Formal call (calling a function with formal parameter names)
 - Informal call (calling a function without formal parameter names)

Formal Call

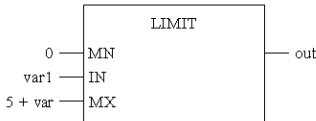
With formal calls (calls with formal parameter names), the call consists of the actual parameter of the output, followed by the assignment instruction `:=`, then the function name and then by a bracketed list of value assignments (actual parameters) to the formal parameter. The order in which the formal parameters are enumerated in a function call is **not significant**.

`EN` and `ENO` can be used for this type of call.

Calling a function with formal parameter names:



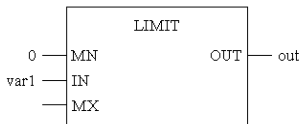
Calling the same function in FBD:



With formal calls it is not necessary to assign a value to all formal parameters, page 443.

```
out:=LIMIT (MN:=0, IN:=var1) ;
```

Calling the same function in FBD:

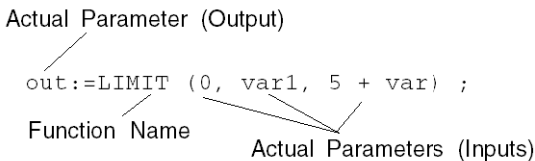


Informal Call

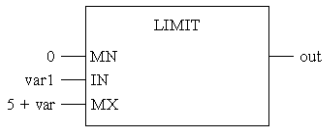
With informal calls (calls without formal parameter names), the call consists of the actual parameter of the output, followed by the symbol of the assignment instruction :=, then the function name and then by a bracketed list of the inputs actual parameters. The order that the actual parameters are enumerated in a function call is **significant**.

EN and ENO **cannot** be used for this type of call.

Calling a function without formal parameter names:



Calling the same function in FBD:



With informal calls it is not necessary to assign a value to all formal parameters, page 443.

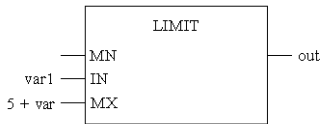
This is a supplement to IEC 61131-3 and must be enabled explicitly.

An empty parameter field is used to skip a parameter.

Call with empty parameter field:

```
out:=LIMIT ( ,var1, 5 + var) ;
```

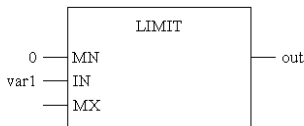
Calling the same function in FBD:



An empty parameter field does not have to be used if formal parameters are omitted at the end.

```
out:=LIMIT (0, var1) ;
```

Calling the same function in FBD:



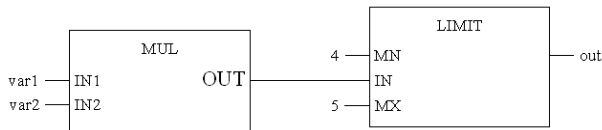
Nesting Functions

A function call can include the call of further functions. The nesting depth is not limited.

Nested call of array function:

```
out:=LIMIT (MN:=4, IN:=MUL (IN1:=var1, IN2:=var2), MX:=5) ;
```

Calling the same function in FBD:



Functions that return a value of the `ANY_ARRAY` data type can **not** be used **within** a function call.

Unauthorized nesting with `ANY_ARRAY`:

```

      ANY_ARRAY
      /
out:=LIMIT (MN:=4,  IN:=EXAMP (IN1:=var1,  IN2:=var2),  MX:=5) ;
  
```

`ANY_ARRAY` is permitted as the return value of the function called or as a parameter of the nested functions.

Authorized nesting with `ANY_ARRAY`:

```

ANY_ARRAY          ANY_ARRAY          ANY_ARRAY
 /                /                    /
out:=EXAMP (MN:=4, IN:=EXAMP (IN1:=var1, IN2:=var2), MX:=var3)
  
```

EN and ENO

With all functions an `EN` input and an `ENO` output can be configured.

If the value of `EN` is equal to "0", when the function is called, the algorithms defined by the function are not executed and `ENO` is set to "0".

If the value of `EN` is equal to "1", when the function is called, the algorithms which are defined by the function are executed. After successful execution of these algorithms, the value of `ENO` is set to "1". If an error occurs during execution of these algorithms, `ENO` will be set to "0".

If the `EN` pin is not assigned a value, when the FFB is invoked, the algorithm defined by the FFB is executed (same as if `EN` equals to "1").

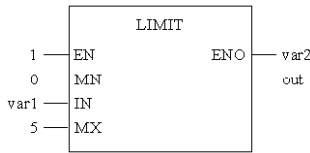
If `ENO` is set to "0" (caused when `EN=0` or an error occurred during executing), the output of the function is set to "0".

The output behavior of the function does not depend on whether the function was called up without `EN/ENO` or with `EN=1`.

If `EN/ENO` are used, the function call must be formal.

```
out:=LIMIT (EN:=1, MN:=0, IN:=var1, MX:=5, ENO=>var2) ;
```

Calling the same function in FBD:



Call Elementary Function Block and Derived Function Block

Elementary Function Block

Elementary function blocks have internal states. If the inputs have the same values, the value on the output can have another value during the individual operations. For example, with a counter, the value on the output is incremented.

Function blocks can have several output values (outputs).

Derived Function Block

Derived function blocks (DFBs) have the same characteristics as elementary function blocks. The user can create them in the programming languages FBD, LD, IL, and/or ST.

Parameters

"Inputs and outputs" are required to transfer values to or from function blocks. These are called formal parameters.

The current process states are transferred to the formal parameters. They are called actual parameters.

The following can be used as actual parameters for function block inputs:

- Variable
- Address
- Literal

The following can be used as actual parameters for function block outputs:

- Variable
- Address

The data type of the actual parameters must match the data type of the formal parameters. The only exceptions are generic formal parameters whose data type is determined by the actual parameter.

When dealing with generic `ANY_BIT` formal parameters, actual parameters of the `INT` or `DINT` (not `UINT` and `UDINT`) data types can be used.

This is a supplement to IEC 61131-3 and must be enabled explicitly.

Example:

Allowed:

```
AND (AnyBitParam := IntVar1, AnyBitParam2 := IntVar2);
```

Not allowed:

```
AND_WORD (WordParam1 := IntVar1, WordParam2 := IntVar2);
```

(In this case, `AND_INT` must be used.)

```
AND_ARRAY_WORD (ArrayInt, ...);
```

(In this case an explicit type conversion must be carried out using `INT_ARR_TO_WORD_ARR (...);.`)

Not all formal parameters must be assigned with a value. The formal parameter types that must be assigned a value are in the following table:

| Parameter type | EDT | STRING | ARRAY | ANY_ARRAY | IODDT | Device DDT | STRUCT | FB | ANY |
|--|-----|--------|-------|-----------|-------|------------|--------|----|-----|
| EFB: Input | - | - | - | - | / | / | - | / | - |
| EFB: VAR_IN_OUT | + | + | + | + | + | / | + | / | + |
| EFB: Output | - | - | + | + | + | / | - | / | + |
| DFB: Input | - | - | - | - | / | + | - | / | - |
| DFB: VAR_IN_OUT | + | + | + | + | + | + | + | / | + |
| DFB: Output | - | - | + | / | / | / | - | / | + |
| + Actual parameter required | | | | | | | | | |
| - Actual parameter not required, it's the general rule, but there are exceptions for some FFBs, for instance when some parameters are used to characterize the information we want to be given by the FFB. | | | | | | | | | |
| / not applicable | | | | | | | | | |

If no value is allocated to a formal parameter, then the initial value will be used for executing the function block. If no initial value has been defined then the default value (0) is used.

If a formal parameter is not assigned with a value and the function block/DFB is instanced more than once, then the following instances are run with the old value.

NOTE: An ANY_ARRAY_xxx input pin not connected will create automatically an hidden array of 1 element.

Public Variables

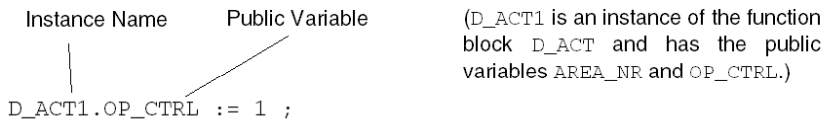
In addition to inputs and outputs, some function blocks also provide public variables.

These variables transfer statistical values (values that are not influenced by the process) to the function block. They are used for setting parameters for the function block.

Public variables are a supplement to IEC 61131-3.

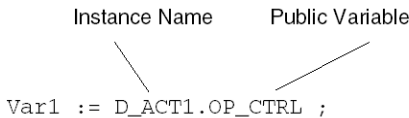
The assignment of values to public variables is made via their initial values or assignments.

Example:



Public variables are read via the instance name of the function block and the names of the public variables.

Example:



Private Variables

In addition to inputs, outputs and public variables, some function blocks also provide private variables.

Like public variables, private variables are used to transfer statistical values (values that are not influenced by the process) to the function block.

Private variables can not be accessed by user program. These type of variables can only be accessed by the animation table.

NOTE: Nested DFBs are declared as private variables of the parent DFB. So their variables are also not accessible through programming, but through the animation table.

Private variables are a supplement to IEC 61131-3.

Programming Notes

Attention should be paid to the following programming notes:

- Functions blocks are only executed if the input EN = 1 or is not used, page 455.
- The assignment of variables to ANY or ARRAY output types must be made using the => operator.

Assignments cannot be made outside of the function block call.

The instruction

```
My_Var := My_SAH.OUT;
```

is **invalid**, if the output OUT of the SAH function block is of type ANY.

The instruction

```
Cal My_SAH (OUT=>My_Var);
```

is **valid**.

- Special conditions apply when using VAR_IN_OUT variables, page 455.
- The use of function blocks consists of two parts in ST:
 - Declaration, page 451
 - calling the function block
- There are two ways of calling a function block:
 - Formal call, page 451 (calling a function with formal parameter names)
This way variables can be assigned to outputs using the => operator.
 - Informal call, page 452 (call without formal parameter names)
- Function block/DFB instances can be called multiple times, page 454; other than instances of communication EFBs, which can only be called once.

Declaration

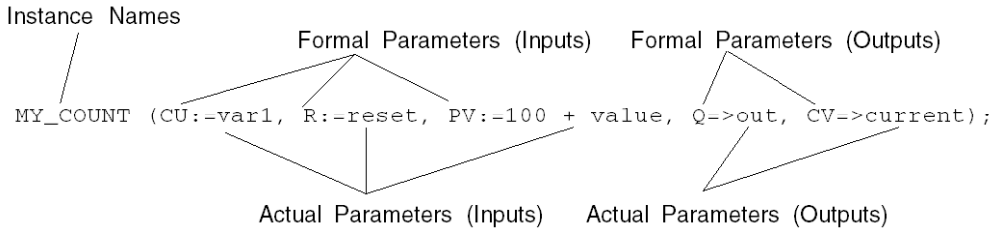
Before calling a function block it must be declared in the variables editor.

Formal Call

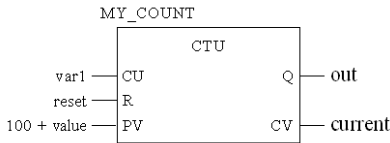
With formal calls (calls with formal parameter names), the function block is called using an instruction sequence made from the function blocks instance names followed a bracketed list of actual parameter assignments to the formal parameters. Assign input formal parameters using := operator, and for output formal parameters using the => operator. The sequence in which the input formal parameters and output formal parameters are enumerated is **not significant**.

EN and ENO can be used for this type of call.

Calling a function block with formal parameter names:



Calling the same function block in FBD:



Assigning the value of a function block output is made by entering the actual parameter name, followed by the assignment instruction := followed by the instance name of the function block and loading the formal parameter of the function block output (separated by a full-stop).

For example,

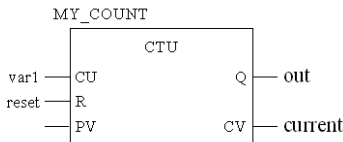
```
MY_COUNT (CU:=var1, R:=reset, PV:=100 + value);
Q := MY_COUNT.out ;
CV := MY_COUNT.current ;
```

NOTE: Type Array DDTs cannot be assigned this way. However, Type Structure DDTs may be assigned.

It is not necessary to assign a value to all formal Parameters, page 448).

```
MY_COUNT (CU:=var1, R:=reset, Q=>out, CV=>current);
```

Calling the same function block in FBD:



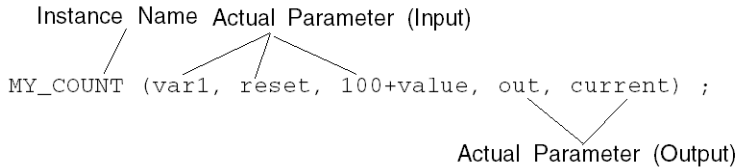
Informal Call

With informal calls (call without Formal parameter names), the function block is called using an instruction made from the function block instance names, followed by a bracketed list of

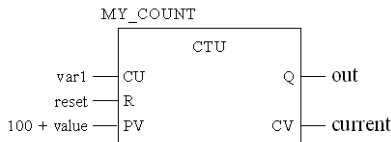
the actual parameters for the inputs and outputs. The order in which the actual parameters are listed in a function block call **is significant**.

EN and ENO **cannot** be used for this type of call.

Calling a function block without formal parameter names:



Calling the same function block in FBD:



With informal calls it is not necessary to assign a value to all formal Parameters, page 448). This does not apply for VAR_IN_OUT variables, for input parameters with dynamic lengths and outputs of type ANY. It must always be assigned a variable.

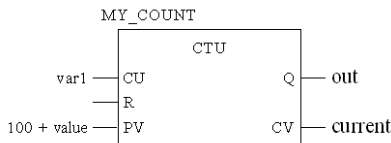
This is a supplement to IEC 61131-3 and must be enabled explicitly.

An empty parameter field is used to skip a parameter.

Call with empty parameter field:

```
MY_COUNT (var1, , 100 + value, out, current) ;
```

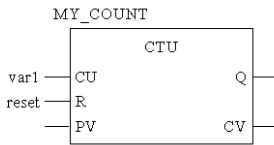
Calling the same function block in FBD:



An empty parameter field does not have to be used if formal parameters are omitted at the end.

```
MY_COUNT (var1, reset) ;
```

Calling the same function block in FBD:



Calling a Function Block without Inputs

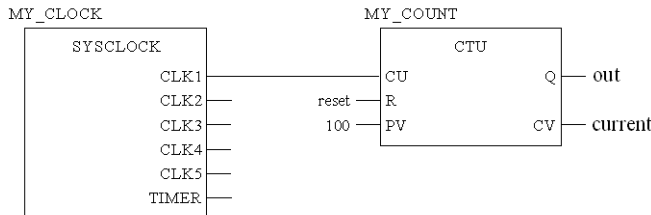
Even if the function block has no inputs or the inputs are not to be parameterized, the function block should be called before its outputs can be used. Otherwise the initial values of the outputs will be transferred, i.e. "0".

For example:

Calling the function block in ST:

```
MY_CLOCK () ;MY_COUNT (CU:=MY_CLOCK.CLK1, R:=reset, PV:=100,
                        Q=>out, CV=>current) ;
```

Calling the same function block in FBD:



Multiple Function Block Instance Call

Function block/DFB instances can be called multiple times; other than instances of communication EFBs, these can only be called once.

Calling the same function block/DFB instance more than once makes sense, for example, in the following cases:

- If the function block/DFB has no internal value or it is not required for further processing.

In this case, memory is saved by calling the same function block/DFB instance more than once since the code for the function block/DFB is only loaded once.

The function block/DFB is then handled like a "Function".

- If the function block/DFB has an internal value and this is supposed to influence various program segments, for example, the value of a counter should be increased in different parts of the program.

In this case, calling the same function block/DFB means that temporary results do not have to be saved for further processing in another part of the program.

EN and ENO

With all function blocks/DFBs, an `EN` input and an `ENO` output can be configured.

If the value of `EN` is equal to "0", when the function block/DFB is called, the algorithms defined by the function block/DFB are not executed and `ENO` is set to "0".

If the value of `EN` is equal to "1", when the function block/DFB is invoked, the algorithms which are defined by the function block/DFB will be executed. After the algorithms have been executed successfully, the value of `ENO` is set to "1". If an error occurred while executing the algorithms, `ENO` is set to "0".

If the `EN` pin is not assigned a value, when the FFB is invoked, the algorithm defined by the FFB is executed (same as if `EN` equals to "1").

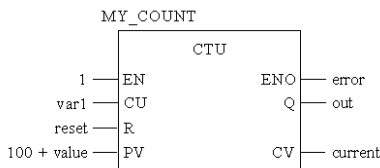
If `ENO` is set to "0" (results from `EN=0` or an error during execution), the outputs of the function block/DFB retain the status from the last cycle in which they were correctly executed.

The output behavior of the function blocks/DFBs does not depend on whether the function blocks/DFBs are called without `EN/ENO` or with `EN=1`.

If `EN/ENO` are used, the function block call must be formal. The assignment of variables to `ENO` must be made using the `=>` operator.

```
MY_COUNT (EN:=1, CU:=var1, R:=reset, PV:=100 + value,
          ENO=>error, Q=>out, CV=>current) ;
```

Calling the same function block in FBD:



VAR_IN_OUT-Variable

Function blocks are often used to read a variable at an input (input variables), to process it and to restate the altered values of the same variable (output variables). This special type of input/output variable is also called a `VAR_IN_OUT` variable.

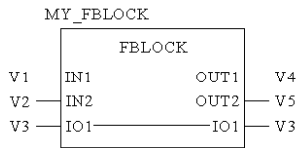
The following special features are to be noted when using function blocks/DFBs with VAR_IN_OUT variables.

- All VAR_IN_OUT inputs must be assigned a variable.
- VAR_IN_OUT inputs may not have literals or constants assigned to them.
- VAR_IN_OUT outputs may **not** have values assigned to them.
- VAR_IN_OUT variables **cannot** be used outside of the function block call.

Calling a function block with VAR_IN_OUT variable in ST:

```
MY_FBLOCK(IN1:=V1, IN2:=V2, IO1:=V3, OUT1=>V4, OUT2=>V5);
```

Calling the same function block in FBD:



VAR_IN_OUT variables **cannot** be used outside the function block call.

The following function block calls are therefore **invalid**:

Invalid call, example 1:

| | |
|---------------------------------|--|
| <pre>InOutFB.inout := V1;</pre> | <p>Assigning the variables V1 to a VAR_IN_OUT parameter.</p> <p>Error: The operation cannot be executed since the VAR_IN_OUT parameter cannot be accessed outside of the function block call.</p> |
|---------------------------------|--|

Invalid call, example 2:

| | |
|---------------------------------|---|
| <pre>V1 := InOutFB.inout;</pre> | <p>Assigning a VAR_IN_OUT parameter to the V1 variable.</p> <p>Error: The operation cannot be executed since the VAR_IN_OUT parameter cannot be accessed outside of the function block call.</p> |
|---------------------------------|---|

The following function block calls are always **valid**:

Valid call, example 1:

| | |
|---------------------------------|---|
| <pre>InOutFB (inout:=V1);</pre> | <p>Calling a function block with the VAR_IN_OUT parameter and formal assignment of the actual parameter within the function block call.</p> |
|---------------------------------|---|

Valid call, example 2:

| | |
|--------------------------|---|
| <pre>InOutFB (V1);</pre> | <p>Calling a function block with the VAR_IN_OUT parameter and informal assignment of the actual parameter within the function block call.</p> |
|--------------------------|---|

Procedures

Procedure

Procedures are provided in the form of libraries. The logic of the procedure is created in the programming language C and may not be modified in the ST editor.

Procedures - like functions - have no internal states. If the input values are the same, the value on the output is the same for all executions of the procedure. For example, the addition of two values gives the same result at every execution.

In contrast to functions, procedures do not return a value and support `VAR_IN_OUT` variables.

Procedures are a supplement to IEC 61131-3 and must be enabled explicitly.

Parameter

"Inputs and outputs" are required to transfer values to or from procedures. These are called formal parameters.

The current process states are transferred to the formal parameters. These are called actual parameters.

The following can be used as actual parameters for procedure inputs:

- Variable
- Address
- Literal
- ST Expression

The following can be used as actual parameters for procedure outputs:

- Variable
- Address

The data type of the actual parameters must match the data type of the formal parameters. The only exceptions are generic formal parameters whose data type is determined by the actual parameter.

When dealing with generic `ANY_BIT` formal parameters, actual parameters of the `INT` or `DINT` (not `UINT` and `UDINT`) data types can be used.

This is a supplement to IEC 61131-3 and must be enabled explicitly.

Example:

Allowed:

```
AND (AnyBitParam := IntVar1, AnyBitParam2 := IntVar2);
```

Not allowed:

```
AND_WORD (WordParam1 := IntVar1, WordParam2 := IntVar2);
```

(In this case, AND_INT must be used.)

```
AND_ARRAY_WORD (ArrayInt, ...);
```

(In this case an explicit type conversion must be carried out using INT_ARR_TO_WORD_ARR (...);).

Not all formal parameters must be assigned with a value. You can see which formal parameter types must be assigned with a value in the following table.

| Parameter type | EDT | STRING | ARRAY | ANY_ARRAY | IODDT | STRUC-T | FB | ANY |
|---------------------------------|-----|--------|-------|-----------|-------|---------|----|-----|
| Input | - | - | + | + | + | + | + | + |
| VAR_IN_OUT | + | + | + | + | + | + | / | + |
| Output | - | - | - | - | - | - | / | + |
| + Actual parameter required | | | | | | | | |
| - Actual parameter not required | | | | | | | | |
| / not applicable | | | | | | | | |

If no value is allocated to a formal parameter, then the initial value will be used for executing the function block. If no initial value has been defined then the default value (0) is used.

Programming Notes

Attention should be paid to the following programming notes:

- Procedures are only executed if the input EN=1 or the EN input is not used (see also EN and ENO, page 460).
- Special conditions apply when using VAR_IN_OUT variables, page 461.
- There are two ways of calling a procedure:
 - Formal call, page 458 (calling a function with formal parameter names)
This way variables can be assigned to outputs using the => operator.
 - Informal call, page 459 (call without formal parameter names)

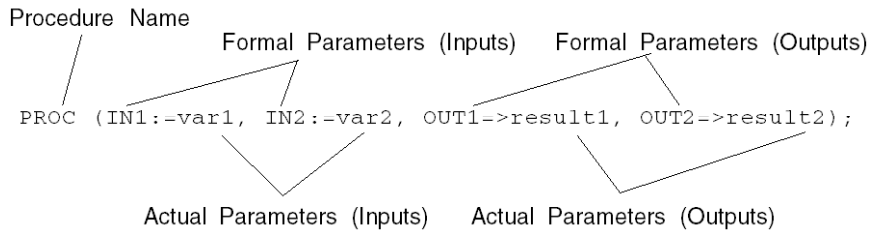
Formal Call

With formal calls (call with formal parameter names), the procedures are called using an instruction sequence made from the procedure name, followed by a bracketed list of actual

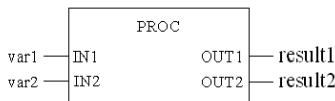
parameter assignments to the formal parameters. The assignment of the input formal parameter is made using the := assignment and the output formal parameter is made using the => assignment. The sequence in which the input formal parameters and output formal parameters are enumerated is **not significant**.

EN and ENO can be used for this type of call.

Calling a procedure with formal parameter names:



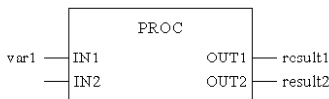
Calling the same procedure in FBD:



With formal calls it is not necessary to assign a value to all formal parameters (see also Parameter, page 457).

```
PROC (IN1:=var1, OUT1=>result1, OUT2=>result2);
```

Calling the same procedure in FBD:

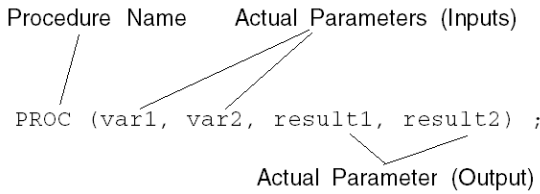


Informal Call

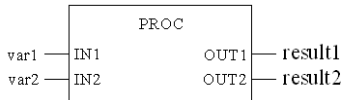
With informal calls (call without formal parameter names), procedures are called using an instruction made from the procedure name, followed by a bracketed list of the inputs and outputs actual parameters. The order that the actual parameters are enumerated in a procedure call **is significant**.

EN and ENO **cannot** be used for this type of call.

Calling a procedure without formal parameter names:



Calling the same procedure in FBD:



With informal calls it is not necessary to assign a value to all formal parameters (see also Parameter, page 457).

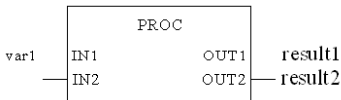
This is a supplement to IEC 61131-3 and must be enabled explicitly.

An empty parameter field is used to skip a parameter.

Call with empty parameter field:

```
PROC (var1, , result1, result2) ;
```

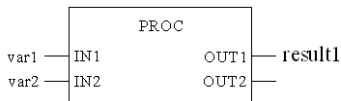
Calling the same procedure in FBD:



An empty parameter field does not have to be used if formal parameters are omitted at the end.

```
PROC (var1, var2, result1) ;
```

Calling the same procedure in FBD:



EN and ENO

With all procedures, an EN input and an ENO output can be configured.

If the value of `EN` is equal to "0", when the procedure is called, the algorithms defined by the procedure are not executed and `ENO` is set to "0".

If the value of `EN` is "1" when the procedure is called, the algorithms defined by the function are executed. After successful execution of these algorithms, the value of `ENO` is set to "1". If an error occurs during execution of these algorithms, `ENO` will be set to "0".

If the `EN` pin is not assigned a value, when the FFB is invoked, the algorithm defined by the FFB is executed (same as if `EN` equals to "1").

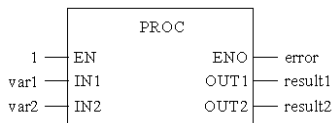
If `ENO` is set to "0" (caused when `EN=0` or an error occurred during executing), the outputs of the procedure are set to "0".

The output behavior of the procedure does not depend on whether the function is called without `EN` or with `EN=1`.

If `EN/ENO` are used, the procedure call must be formal. The assignment of variables to `ENO` must be made using the `=>` operator.

```
PROC (EN:=1, IN1:=var1, IN2:=var2,
      ENO=>error, OUT1=>result1, OUT2=>result2) ;
```

Calling the same procedure in FBD:



VAR_IN_OUT Variable

Procedures are often used to read a variable at an input (input variables), to process it and to restate the altered values of the same variable (output variables). This special type of input/output variable is also called a `VAR_IN_OUT` variable.

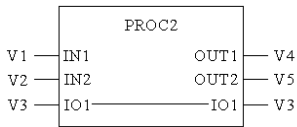
The following special features are to be noted when using procedures with `VAR_IN_OUT` variables.

- All `VAR_IN_OUT` inputs must be assigned a variable.
- `VAR_IN_OUT` inputs may not have literals or constants assigned to them.
- `VAR_IN_OUT` outputs may **not** have values assigned to them.
- `VAR_IN_OUT` variables **cannot** be used outside of the procedure call.

Calling a procedure with `VAR_IN_OUT` variable in ST:

```
PROC2 (IN1:=V1, IN2:=V2, IO1:=V3,
      OUT1=>V4, OUT2=>V5) ;
```

Calling the same procedure in FBD:



VAR_IN_OUT variables **cannot** be used outside of the procedure call.

The following procedure calls are therefore **invalid**:

Invalid call, example 1:

| | |
|-------------------------------------|---|
| <code>InOutProc.inout := V1;</code> | <p>Assigning the variables V1 to a VAR_IN_OUT parameter.</p> <p>Error: The operation cannot be executed since the VAR_IN_OUT parameter cannot be accessed outside of the procedure call.</p> |
|-------------------------------------|---|

Invalid call, example 2:

| | |
|-------------------------------------|--|
| <code>V1 := InOutProc.inout;</code> | <p>Assigning a VAR_IN_OUT parameter to the V1 variable.</p> <p>Error: The operation cannot be executed since the VAR_IN_OUT parameter cannot be accessed outside of the procedure call.</p> |
|-------------------------------------|--|

The following procedure calls are always **valid**:

Valid call, example 1:

| | |
|-------------------------------------|---|
| <code>InOutProc (inout:=V1);</code> | <p>Calling a procedure with the VAR_IN_OUT parameter and formal assignment of the actual parameter within the procedure call.</p> |
|-------------------------------------|---|

Valid call, example 2:

| | |
|------------------------------|---|
| <code>InOutProc (V1);</code> | <p>Calling a procedure with the VAR_IN_OUT parameter and informal assignment of the actual parameter within the procedure call.</p> |
|------------------------------|---|

User Function Blocks (DFB)

What's in This Part

| | |
|---|-----|
| Overview of User Function Blocks (DFB) | 464 |
| Description of User Function Blocks (DFB) | 469 |
| User Function Blocks (DFB) Instance | 478 |
| Use of the DFBs from the Different Programming Languages | 484 |
| User Diagnostics DFB | 501 |
| Implicit Type Conversion in Control Expert | 503 |

In This Part

This part presents:

- The user function blocks (DFB)
- The internal structure of DFBs
- Diagnostics DFBs
- The types and instances of DFBs
- The instance calls using different languages

Overview of User Function Blocks (DFB)

What’s in This Chapter

| | |
|--|-----|
| Introduction to User Function Blocks | 464 |
| Implementing a DFB Function Block..... | 466 |

Subject of this Chapter

This chapter provides an overview of the user function blocks (DFB), and the different steps in their implementation.

Introduction to User Function Blocks

Introduction

Control Expert software enables you to create DFB user function blocks, using automation languages. A DFB is a program block that you write to meet the specific requirements of your application. It includes:

- one or more sections written in Ladder (LD), Instruction List (IL), Structured Text (ST) or Functional Block Diagram (FBD) language
- input/output parameters
- public or private internal variables

Function blocks can be used to structure and optimize your application. They can be used whenever a program sequence is repeated several times in your application, or to set a standard programming operation (for example, an algorithm that controls a motor, incorporating local safety requirements).

By exporting then importing these blocks, they can be used by a group of programmers working on a single application or in different applications.

Benefits of Using a DFB

Using a DFB function block in an application enables you to:

- simplify the design and entry of the program
- increase the legibility of the program
- facilitate the debugging of the application (all of the variables handled by the function block are identified on its interface)

- reduce the volume of code generated (the code that corresponds to the DFB is only loaded once - however many calls are made to the DFB in the program, only the data corresponding to the instances are generated)

Comparison with a Subroutine

Compared to a subroutine, using a DFB makes it possible to:

- set processing parameters more easily
- use internal variables that are specific to the DFB and therefore independent from the application
- test its operation independently from the application

Furthermore, LD and FBD languages provide a graphic view of the DFBs, facilitating the design and debugging of your program.

DFB Created with Previous Software Versions

DFBs created using PL7 and Concept must first be converted using the converters that come with the product, before being used in the application.

Domain of Use

The following table shows the domain of use for the DFBs.

| Function | Domain |
|---|----------------------------|
| PLCs for which DFBs can be used. | Premium\Atrium and Quantum |
| DFB creation software | Control Expert |
| Software with which DFBs can be used. | Control Expert |
| Programming language for creating the DFB code. | IL, ST, LD or FBD (1) |
| Programming language with which DFBs can be used. | IL, ST, LD or FBD (1) |

(1) IL: Instruction List , ST: Structured Text, LD: LaDder, FBD: Functional Block Diagram language.

Implementing a DFB Function Block

Implementation Procedure

There are 3 steps in the DFB function block implementation procedure:

| Step | Action |
|------|--|
| 1 | Create your DFB model (called: DFB type). |
| 2 | Create a copy of this function block, called an instance, every time the DFB is used in the application. |
| 3 | Use the DFB instances in your application program. |

Creation of the DFB Type

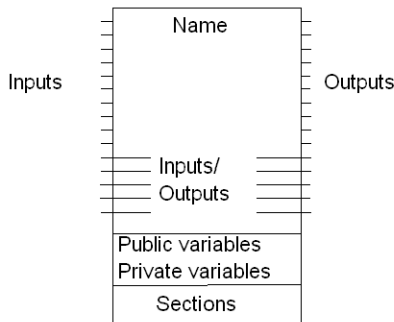
This operation consists in designing a model of the DFB you want to use in your application. To do this, use the DFB editor to define and code all the elements that make up the DFB:

- Description of the function block: name, type (DFB), activation of diagnostics, comment.
- Structure of the function block: parameters, variables, code sections.

NOTE: If you use a DFB that is already in the User-Defined Library and modify it, the new modified type will be used for any additional instances in the open project. However, the User-Defined Library remains unchanged.

Description of a DFB Type

The following diagram shows a graphic representation of a DFB model.



The function block comprises the following elements:

- Name: name of the DFB type (max. 32 characters). This name must be unique in the libraries, the authorized characters used depend on the choice made in the **Identifiers**

area of the **Language extensions** tab in the **Project Settings** (see EcoStruxure™ Control Expert, Operating Modes):

- Inputs: input parameters (excluding input/output parameters).
- Outputs: output parameters (excluding input/output parameters).
- Inputs/Outputs: input/output parameters.
- Public variables: internal variables accessible by the application program.
- Private variables: nested internal variables or DFBs, not accessible by the application program.
- Sections: DFB code sections in LD, IL, ST or FBD.
- Comment of a maximum of 1024 characters. Formatting characters (carriage return, tab, etc.) are not authorized.

For each type of DFB, a descriptive file is also accessible via a dialog box: size of the DFB, number of parameters and variables, version number, date of last modification, protection level, etc.

Online Help for DFB Types

It is possible to link an HTML help file to each DFB in the User-Defined Library. This file must:

- Have a name that is identical to the linked DFB,
- Be located in the directory `\Schneider Electric\FFBLibset\CustomLib\MyCustomFam\Language` (where **Language** is named **Eng**, **Fre**, **Ger**, **Ita**, **Spa** or **Chs** according to the language desired).

Creation of a DFB Instance

Once the DFB type is created, you can define an instance of this DFB via the variable editor or when the function is called in the program editor.

Use of DFB Instances

A DFB instance is used as follows

- as a standard function block in Ladder (LD) or Functional Block Diagram (FBD) language,
- as an elementary function in Structured Text (ST) or Instruction List (IL) language.

A DFB instance can be used in all application program tasks, except event tasks and Sequential Function Chart (SFC) transitions.

Storage

The DFB types the user creates can be stored (see EcoStruxure™ Control Expert, Operating Modes) in the function and function block library.

Description of User Function Blocks (DFB)

What's in This Chapter

| | |
|--|-----|
| Definition of DFB Function Block Internal Data | 469 |
| DFB Parameters | 471 |
| DFB Variables | 475 |
| DFB Code Section | 476 |

Subject of this Chapter

This chapter provides an overview of the different elements that make up the user function blocks.

Definition of DFB Function Block Internal Data

At a Glance

There are two types of DFB internal data:

- The parameters: Input, Output or Input/Output.
- Public or Private variables.

The internal data of the DFB must be defined using symbols (this data cannot be addressed as an address).

Elements to Define for Each Parameter

When the function block is created, the following must be defined for each parameter:

- Name: Name of DFB type (max. 32 characters). This name must be unique in the libraries; the authorized characters used depend on the choice made in the **Identifiers** area of the **Language extensions** tab in **Project Settings** (see EcoStruxure™ Control Expert, Operating Modes):
- A type of object (BOOL, INT, REAL, etc.).
- A comment of a maximum of 1024 characters (optional). Formatting characters (carriage return, tab, etc.) are not allowed.
- An initial value.
- The read/write attribute that defines whether the variable may or may not be written in runtime: R (read only) or R/W (read/write). This attribute must only be defined for public variables.

- The backup attribute that defines whether the variable may or may not be saved.

Types of Objects

The types of objects that may be defined for the DFB parameters belong to the following families:

- Elementary data family: EDT. This family includes the following object types: Boolean (BOOL, EBOOL), Integer (INT, DINT, etc.), Real (REAL), Character string (STRING), Bit string (BYTE, WORD, etc.), etc.
- Derived data family: DDT. This family includes table (ARRAY) and structure (user or IODDT) object types.
- Generic data families: ANY_ARRAY_xxx.
- The function block family: FB. This family includes EFB and DFB object types.

Authorized Objects for the Different Parameters

For performances reasons, the addressing mode of the DFB parameters must be transferred by address for the following object families

- Inputs
- Inputs/Outputs
- Outputs

The addressing mode of a Function Block element is linked to the element type. The addressing modes are passed by:

- Value (VAL)
- Relocation table entry (RTE)
- Logical address: RTE+Offset (L-ADR)
- Logical address and number of elements (L-ADR-LG)
- IO channel structure (IOCHS)

For each of the DFB parameters, the following object families may be used with its associated addressing modes:

| Object families | EDT | STRING | Anonymous or DDT array | DDT ⁽¹⁾ | IODD-T | GDT: ANY_ARRAY_x | FB | ANY... |
|-----------------|----------------------|----------|------------------------|--------------------|------------------|------------------|----|----------|
| Inputs | VAL | L-ADR-LG | L-ADR-LG | L-ADR | No | L-ADR-LG | No | L-ADR-LG |
| Inputs/outputs | L-ADR ⁽²⁾ | L-ADR-LG | L-ADR-LG | L-ADR | IOCH-S, page 487 | L-ADR-LG | No | L-ADR-LG |

| Object families | EDT | STRING | Anonymous or DDT array | DDT ⁽¹⁾ | IODD-T | GDT: ANY_ARRAY_x | FB | ANY... |
|-------------------|--|--------|------------------------|--------------------|--------|------------------|-----|----------|
| Outputs | VAL | VAL | L-ADR-LG | VAL | No | L-ADR-LG | No | L-ADR-LG |
| Public variables | VAL | VAL | VAL | VAL | No | No | No | No |
| Private variables | VAL | VAL | VAL | VAL | No | No | RTE | No |
| Key: | | | | | | | | |
| (1) | Derived data family, except input/output derived data types (IODDT). | | | | | | | |
| (2) | Except for EBOOL-type static variables, with Quantum PLCs. | | | | | | | |

⚠ CAUTION

UNEXPECTED APPLICATION BEHAVIOR - ARRAY INDEX

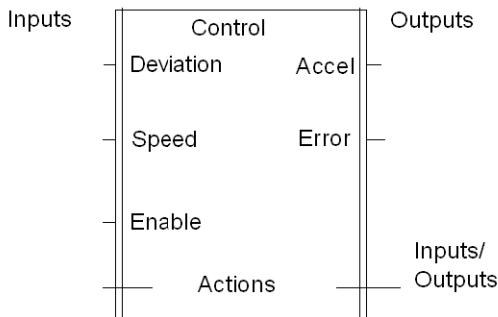
Take into account the shift of the index for ARRAY variables that have a not null start index on ANY_ARRAY_x entry (the shift equals the start index value).

Failure to follow these instructions can result in injury or equipment damage.

DFB Parameters

Illustration

This illustration shows some examples of DFB parameters



Description of the Parameters

This table shows the role of each parameter

| Parameter | Maximum number | Role |
|----------------|----------------|---|
| Inputs | 32 (1) | These parameters can be used to transfer the values of the application program to the internal program of the DFB. They are accessible in read-only by the DFB, but are not accessible by the application program. |
| Outputs | 32 (2) | These parameters can be used to transfer the values of the DFB to the application program. They are accessible for reading by the application program except for ARRAY-type parameters. |
| Inputs/Outputs | 32 | These parameters may be used to transfer data from the application program to the DFB, which can then modify it and return it to the application program. These parameters are not accessible by the application program. |

Legend:

(1) Number of inputs + Number of inputs/outputs less than or equal to 32

(2) Number of outputs + Number of inputs/outputs less than or equal to 32

NOTE: The IODDT related to CANopen devices for Modicon M340 cannot be used as a DFB I/O parameter. During the analyse/build step of a project, the following message: "This IODDT cannot be used as a DFB parameter" advises the limitations to the user.

Parameters that Can Be Accessed by the Application Program

The only parameters that can be accessed by the application program outside the call are output parameters. To make this possible, the following syntax must be used in the program:

DFB_Name.Parameter_name

DFB_Name represents the name of the instance of the DFB used (maximum of 32 characters).

Parameter_Name represents the name of the output parameter (maximum 32 characters).

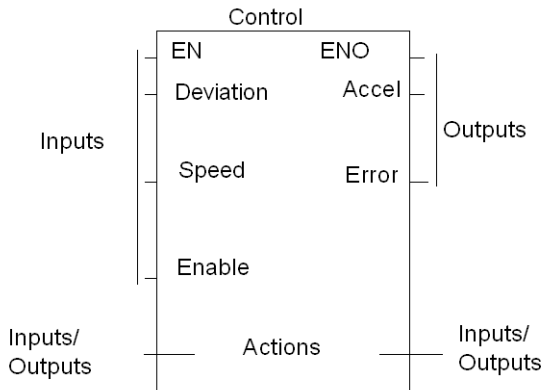
Example: `Control.Accel` indicates the output `Accel` of the DFB instance called `Control`

EN and ENO Parameters

EN is an input parameter, and **ENO** is an output parameter. They are both of BOOL type, and may or may not be used (optional) in the definition of a DFB type.

Where the user wishes to use these parameters, the editor sets them automatically: EN is the first input parameter and ENO the first output parameter.

Example of implementation of EN\ENO parameters.



If the EN input parameter of an instance is assigned the value 0 (FALSE), then:

- the section(s) that make up the code of the DFB is/are not executed (this is managed by the system),
- the ENO output parameter is set to 0 (FALSE) by the system.

If the EN input parameter of an instance is assigned the value 1 (TRUE), then:

- the section(s) that make up the code of the DFB is/are executed (this is managed by the system),
- the ENO output parameter is set to 1 (TRUE) by the system.

If an error is detected (for example a processing error) by the DFB instance, the user has the option of setting the ENO output parameter to 0 (FALSE). In this case:

- either the output parameters are frozen in the state they were in during the previous process until the fault disappears,
- or the user provides a function in the DFB code whereby the outputs are forced to the required state until the fault disappears.

VAR_IN_OUT Variable

Function blocks are often used to read a variable at an input (input variables), to process it and to output the updated values of the same variable (output variables). This special type of input/output variable is also called a VAR_IN_OUT variable.

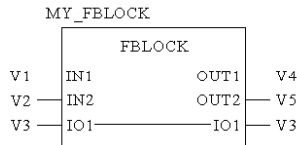
The following special features are to be noted when using function blocks/DFBs with VAR_IN_OUT variables.

- All VAR_IN_OUT inputs must be assigned a variable.
- VAR_IN_OUT inputs may not have literals or constants assigned to them.
- VAR_IN_OUT outputs may **not** have values assigned to them.
- VAR_IN_OUT variables **cannot** be used outside the block call.

Calling a function block with a VAR_IN_OUT variable in IL:

```
CAL MY_FBLOCK(IN1:=V1, IN2:=V2, IO1:=V3,
              OUT1=>V4, OUT2=>V5)
```

Calling the same function block in FBD:



VAR_IN_OUT variables **cannot** be used outside the function block call.

The following function block calls are therefore **invalid**:

Invalid call, example 1:

| | |
|-------------|--|
| LD V1 | Loading a v1 variable in the accumulator |
| CAL InOutFB | Calling a function block with the VAR_IN_OUT parameter. The accumulator now contains a reference to a VAR_IN_OUT parameter. |
| AND V2 | AND operation on accumulator contents and v2 variable. Error: The operation cannot be performed since the VAR_IN_OUT parameter (accumulator contents) cannot be accessed from outside the function block call. |

Invalid call, example 2:

| | |
|-------------------|---|
| LD V1 | Loading a v1 variable in the accumulator |
| AND InOutFB.inout | AND operation on accumulator contents and a reference to a VAR_IN_OUT parameter. Error: The operation cannot be performed since the VAR_IN_OUT parameter cannot be accessed from outside the function block call. |

The following function block calls are always **valid**:

Valid call, example 1:

| | |
|----------------------------------|---|
| CAL InOutFB (IN1:=V1, inout:=V2) | Calling a function block with the VAR_IN_OUT parameter and assigning the actual parameter within the function block call. |
|----------------------------------|---|

Valid call, example 2:

| | |
|------------------------|--|
| LD V1 | Loading a V1 variable in the accumulator |
| ST InOutFB.IN1 | Assigning the accumulator contents to the IN1 parameter of the IN1 function block. |
| CAL InOutFB(inout:=V2) | Calling the function block with assignment of the actual parameter (V2) to the VAR_IN_OUT parameter. |

DFB Variables

Description of the Variables

This table shows the role of each type of variable.

| Variable | Maximum number | Role |
|----------|----------------|--|
| Public | unlimited | These internal variables of the DFB may be used by the DFB, by the application program and by the user in adjust mode. |
| Private | unlimited | These internal variables of the DFB can only be used by this function block, and are therefore not accessible by the application program, but these type of variables can be accessed by the data editor and the animation table. These variables are generally necessary to the programming of the block, but are of no interest to the user (for example, the result of an intermediate calculation, etc.). |

NOTE: Nested DFBs are declared as private variables of the parent DFB. So their variables are also not accessible through programming, but through the data editor and the animation table.

Variables that Can Be Accessed by the Application Program

The only variables that can be accessed by the application program are public variables. To make this possible, the following syntax must be used in the program: **DFB_Name**.
Variable_Name

DFB_Name represents the name of the instance of the DFB used (maximum of 32 characters),

Variable_Name represents the name of the public variable (maximum of 8 characters).

Example: `Control.Gain` indicates the public variable `Gain` of the DFB instance called `Control`

Saving Public Variables

Setting the `%S94` system bit to 1 causes the public variables you have modified to be saved by program or by adjustment, in place of the initial values of these variables (defined in the DFB instances).

Replacement is only possible if the backup attribute is correctly set for the variable.

NOTICE

APPLICATION UPLOAD NOT SUCCESSFUL

The bit `%S94` must not be set to 1 during an upload. If the bit `%S94` is set to 1 upload then the upload may be impossible.

Failure to follow these instructions can result in equipment damage.

DFB Code Section

General

The code section(s) define(s) the process the DFB is to carry out, as a function of the declared parameters.

A DFB may contain several code sections; the number of sections being unlimited.

Programming Languages

To program DFB sections, you can use the following languages:

- Instruction List (IL)
- Structured Text (ST)
- Ladder language (LD)
- Functional Block Diagram (FBD)

Defining a Section

A section is defined by:

- a symbolic name that identifies the section (maximum of 32 characters)
- a validation condition that defines the execution of the section
- a comment (maximum of 256 characters)
- a protection attribute (no protection, write-protected section, read/write-protected section)

Programming Rules

When executed, a DFB section can only use the parameters you have defined for the function block (input, output and input/output parameters and internal variables).

Consequently, a DFB function block cannot use either the global variables of the application, or the input/output objects, except the system words and bits (%Si, %SWi and %SDi).

A DFB section has maximum access rights (read and write) for its parameters.

Example of Code

The following program provides an example of Structured Text code

```

CHR_200:=CHR_100;
CHR_114:=CHR_104;
CHR_116:=CHR_106;
RESET DEMARRE;
(*We increment 80 times CHR_100*)
FOR CHR_102:=1 TO 80 DO
    INC CHR_100;
    WHILE ((CHR_104-CHR_114)<100) DO
        IF (CHR_104>400) THEN
EXIT;
            END IF;
            INC CHR_104;
            REPEAT
                IF (CHR_106>300) THEN
EXIT;
                    END IF;
                    INC CHR_106;
                    UNTIL ((CHR_100-CHR_116)>100)
                    END REPEAT;
            END WHILE;
            (* Loop as long as CHR_106)
            IF (CHR_106=CHR_116)
            THEN EXIT;
            ELSE
                CHR_114:=CHR_104;
                CHR_116:=CHR_106;
            END IF;
            INC CHR_200;
        END FOR;

```

User Function Blocks (DFB) Instance

What's in This Chapter

| | |
|---|-----|
| Creation of a DFB Instance..... | 478 |
| Execution of a DFB Instance..... | 479 |
| Programming Example for a Derived Function Block (DFB)..... | 480 |

Subject of this Chapter

This chapter provides an overview of the creation of a DFB instance, and its execution.

Creation of a DFB Instance

DFB Instance

A DFB instance is a copy of the DFB model (DFB type):

- It uses the DFB type code (the code is not duplicated).
- It creates a data zone specific to this instance, which is a copy of the parameters and variables of the DFB type. This zone is situated in the application's data area.

You must identify each DFB instance you create with a name of a maximum 32 characters, the authorized characters used depend on the choice made in the **Identifiers** area of the **Language extensions** tab in the **Project Settings** (see EcoStruxure™ Control Expert, Operating Modes).

The first character must be a letter! Keywords and symbols are prohibited.

Creation of an Instance

From a DFB type, you can create as many instances as necessary; the only limitation is the size of the PLC memory.

Initial Values

The initial values of the parameters and public variables that you defined when creating the DFB type can be modified for each DFB instance.

Not all DFB parameters have an initial value.

Modification of the initial values of the elements in the DFB instances

| | EDT (except String type) | String Type | EDT | DDT structure | FB | ANY_ARRAY | IODDT | ANY_... |
|--------------------------|--------------------------|-------------|-----|---------------|----|-----------|-------|---------|
| Inputs | Yes | No | No | No | - | No | - | No |
| Input/Output | No | No | No | No | - | No | No | No |
| Outputs | Yes | Yes | No | Yes | - | - | - | No |
| Public variables | Yes | Yes | Yes | Yes | - | - | - | - |
| Private Variables | Yes | Yes | Yes | Yes | No | - | - | - |

Modification of the initial values of the elements in the DFB type

| | EDT (except String type) | String Type | EDT | DDT structure | FB | ANY_ARRAY | IODDT | ANY_... |
|--------------------------|--------------------------|-------------|-----|---------------|----|-----------|-------|---------|
| Inputs | Yes | No | No | No | - | No | - | No |
| Input/Output | No | No | No | No | - | No | No | No |
| Outputs | Yes | Yes | No | Yes | - | - | - | No |
| Public variables | Yes | Yes | Yes | Yes | - | - | - | - |
| Private Variables | Yes | Yes | Yes | Yes | No | - | - | - |

Execution of a DFB Instance

Operation

A DFB instance is executed as follows.

| Step | Action |
|------|--|
| 1 | Loading the values in the input and input/output parameters. On initialization (or on cold restart), all non-assigned inputs take the initial value defined in the DFB type. They then keep the last value assigned to them. |
| 2 | Execution of the internal program of the DFB. |
| 3 | Writing the output parameters. |

NOTE: The internal variables of DFBs are not reinitialized when using **Build project online** command after an input modification. To reinitialize all internal variables use **Rebuild all project** command.

Debugging of DFBs

The Control Expert software offers several DFB debugging tools:

- animation table: all parameters, and public and private variables are displayed and animated in real-time. Objects may be modified and forced
- breakpoint, step by step and program diagnostics
- runtime screens: for unitary debugging

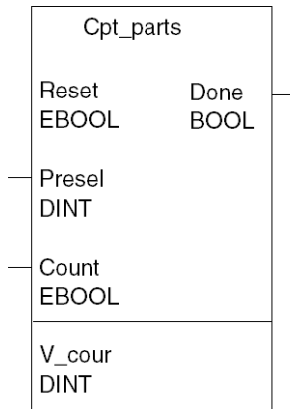
Programming Example for a Derived Function Block (DFB)

General

This example of programming a counter using a DFB is provided for instruction purposes.

Characteristics of the DFB Type

The DFB type used to create the counter is as follows.



The elements of the `Cpt_parts` DFB type are as follows.

| Elements | Description |
|----------------------|--|
| Name of the DFB type | Cpt_parts |
| Input parameters | <ul style="list-style-type: none"> • Reset: counter reset (EBOOL type) |

| Elements | Description |
|--------------------------|--|
| | <ul style="list-style-type: none"> Presel: Preset value of the counter (DINT type) Count: upcounter input (EBOOL type) |
| Output parameters | Done : preset value reached output (BOOL type) |
| Public internal variable | V_cour : current value of the counter (DINT type) |

Operation of the Counter

The operation of the counter must be as follows.

| Phase | Description |
|-------|---|
| 1 | The DFB counts the rising edges on the Count input. |
| 2 | The number of edges it counts is then stored by the variable <code>V_cour</code> . This variable is reset by a rising edge on the <code>Reset</code> input. |
| 3 | When the number of edges counted is equal to the preset value, the <code>Done</code> output is set to 1. This variable is reset by a rising edge on the <code>Reset</code> input. |

Internal Program of the DFB

The internal program of the DFB type `Cpt_parts` is defined in Structured Text as follows.

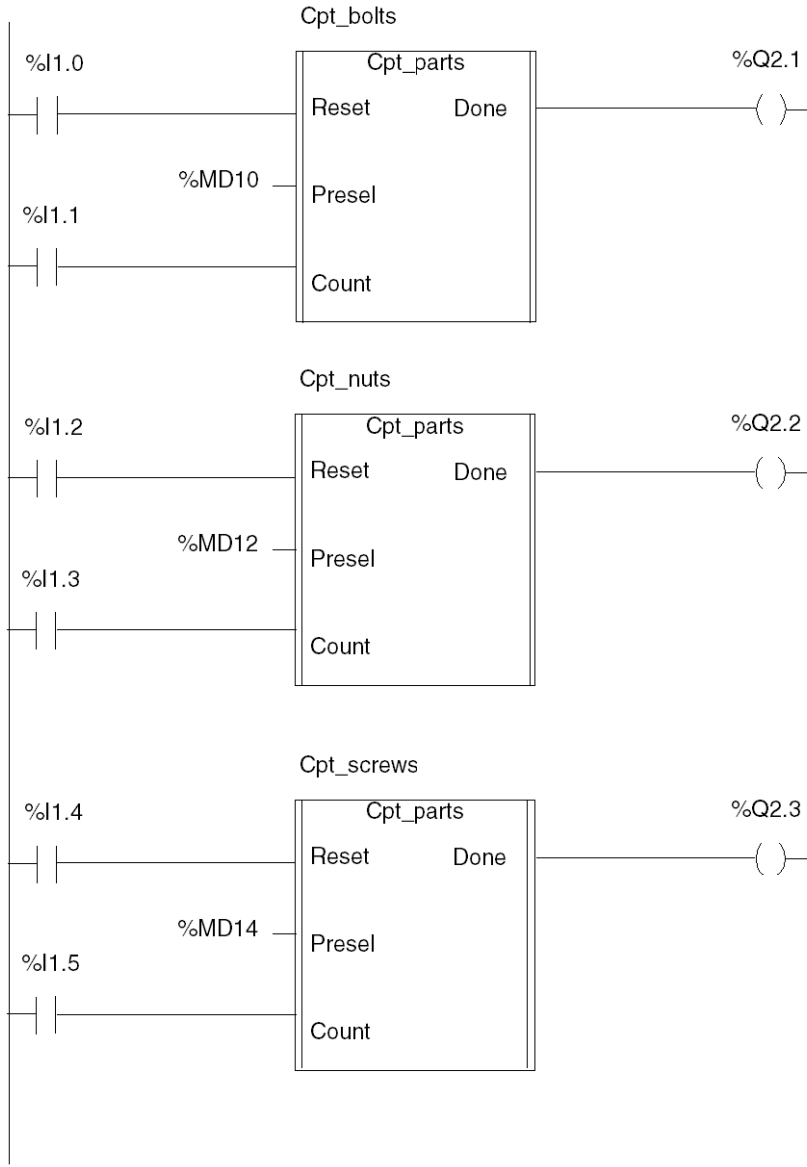
```
!(*Programming of the Cpt_parts DFB*)
IF RE (Reset) THEN
    V_cour:=0;
END_IF;
IF RE (Count) THEN
    V_cour:=V_cour+1;
END_IF;
IF(V_cour>=Presel) THEN
    SET (Done);
ELSE
    RESET (Done);
END_IF;
```

Example of Use

Let us suppose your application needs to count 3 part types (for example, bolts, nuts and screws). The DFB type `Cpt_parts` can be used three times (3 instances) to perform these different counts.

The number of parts to be procured for each type is defined in the words %MD10, %MD12 and %MD14 respectively. When the number of parts is reached, the counter sends a command to an output (%Q1.2.1, %Q1.2.2 or %Q1.2.3) which then stops the procurement system for the corresponding parts.

The application program is entered in Ladder language as follows. The 3 DFBs (instances) Cpt_bolts, Cpt_nuts and Cpt_screws are used to count the different parts.



Use of the DFBs from the Different Programming Languages

What's in This Chapter

| | |
|---|-----|
| Rules for Using DFBs in a Program | 484 |
| Use of IODDTs in a DFB | 487 |
| Use of a DFB in a Ladder Language Program | 490 |
| Use of a DFB in a Structured Text Language Program | 492 |
| Use of a DFB in an Instruction List Program | 495 |
| Use of a DFB in a Program in Function Block Diagram Language..... | 498 |

Subject of this Chapter

This chapter provides an overview of DFB instance calls made using the different programming languages.

Rules for Using DFBs in a Program

General

DFB instances can be used in all languages [Instruction List (IL), Structured Text (ST), Ladder (LD) and Function Block Diagram (FBD)] and in all the tasks of the application program (sections, subroutine, etc.), except for SFC program transition.

General Rules of Use

When using a DFB, you must comply with the following rules for whatever language is being used:

- It is not necessary to connect all the input, input/output or output parameters, except the following parameters, which it is compulsory for you to assign:
 - input/output parameters
 - generic data-type output parameters (ANY_INT, ANY_ARRAY, etc.)
- The following parameters are optional:
 - generic data-type input parameters (ANY_INT, ANY_ARRAY, etc.)
 - STRING-type input parameters

- Unconnected input parameters keep the value of the previous call or the initialization value defined for these parameters, if the block has never been called
- All of the objects assigned to the input, input/output and output parameters must be of the same type as those defined when the DFB type was created (for example: if the type INT is defined for the input parameter "speed", then you cannot assign it the type DINT or REAL)

The only exceptions are BOOL and EBOOL types for input and output parameters (not for input/output parameters), which can be mixed.

Example: The input parameter "Validation" may be defined as BOOL and associated with a %Mi internal bit of type EBOOL. However, in the internal code of the DFB type, the input parameter actually has BOOL-type properties (it cannot manage edges).

Assignment of Parameters

The following table summarizes the different possibilities for assigning parameters in the different programming languages.

| Parameter | Type | Assignment of the parameter (1) | Assignment |
|----------------|------------|--|--------------|
| Inputs | EDT (2) | Connected, value, object or expression | Optional (3) |
| | BOOL | Connected, value, object or expression | Optional |
| | DDT | Connected, value or object | Optional |
| | Device DDT | Connected or object | Compulsory |
| | ANY_... | Connected or object | Optional |
| | ANY_ARRAY | Connected or object | Optional |
| Inputs/outputs | EDT | Connected or object | Compulsory |
| | DDT | Connected or object | Compulsory |
| | Device DDT | Connected or object | Compulsory |
| | IODDT | Connected or object | Compulsory |
| | ANY_... | Connected or object | Compulsory |
| | ANY_ARRAY | Connected or object | Compulsory |
| Outputs | EDT | Connected or object | Optional |
| | DDT | Connected or object | Optional |
| | ANY_... | Connected or object | Compulsory |
| | ANY_ARRAY | Connected or object | Compulsory |

- (1) Connected in Ladder (LD) or Function Block Diagram (FBD) language. Value or object in Instruction List (IL) or Structured Text (ST) language.
- (2) Except BOOL-type parameters
- (3) Except for STRING-type parameters that is compulsory.

Rules when using DFBs with arrays

⚠ WARNING

UNEXPECTED EQUIPMENT OPERATION

Check the size of arrays when copying from source into target arrays using DFBs.

Failure to follow these instructions can result in death, serious injury, or equipment damage.

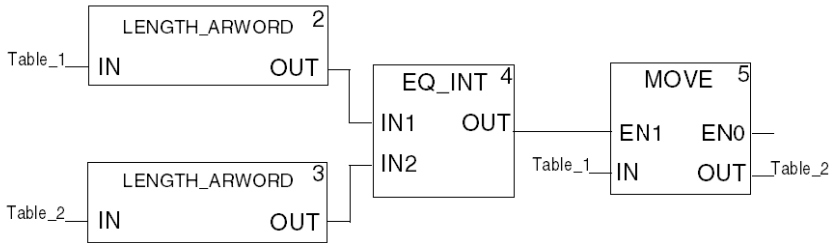
When using dynamic arrays, it is mandatory to check the sizes of arrays that are identical. In specific case, using dynamic arrays as an output or input/output, an overflow could lead to improper execution of the program and stop of the PLC.

This behavior occurs if the following conditions are fulfilled simultaneously:

- Use of a DFB with at least one output or I/O parameter of dynamic array type (`ANY_ARRAY_XXX`).
- In the coding of a DFB, use of a function or function block (FFB of type FIFO, LIFO, MOVE, MVX, T2T, SAH or SEL). Note that, the function or FFB needs two `ANY` type parameters with at least one defined on the output.
- The DFB parameter of the dynamic array is used in writing during the FFB call (on the `ANY` type parameter). For other `ANY` parameters, an array with a fixed size is used.
- The size of the fixed size array is bigger than the size of the dynamic array calculated to store the result.

Example for checking the size of arrays

The following example shows how to check the size of arrays using the function `LENGTH_ARWORD` in a DFB.



In this example, `Table_1` is an array with a fixed size, `Table_2` is a dynamic array of type `ANY_ARRAY_WORD`. This program checks the size of each array. The functions `LENGTH_ARWORD` compute the size of each array in order to condition the execution of the `MOVE` function.

Use of IODDTs in a DFB

At a Glance

The following tables present the different IODDTs for the Modicon M340, Modicon M580, Premium and Quantum PLCs that can be used in a DFB (exclusively as input/output, page 470) parameters.

IODDT that Can Be Used in a DFB

The following table lists the IODDTs of the different application for Modicon M340, Modicon M580, Premium, and Quantum PLCs that can be used in a DFB:

| IODDT families | Modicon M340 | Modicon M580 | Premium | Quantum |
|-----------------------------|--------------|--------------|---------|---------|
| Discrete application | | | | |
| T_DIS_IN_GEN | No | No | No | No |
| T_DIS_IN_STD | No | No | No | No |
| T_DIS_EVT | No | No | No | No |
| T_DIS_OUT_GEN | No | No | No | No |
| T_DIS_OUT_STD | No | No | No | No |

| IODDT families | Modicon M340 | Modicon M580 | Premium | Quantum |
|-----------------------------------|---------------------|---------------------|----------------|----------------|
| T_DIS_OUT_REFLEX | No | No | No | No |
| Analog application | | | | |
| T_ANA_IN_GEN | No | No | No | No |
| T_ANA_IN_STD | No | No | No | No |
| T_ANA_IN_CTRL | No | Yes ^(1.) | Yes | No |
| T_ANA_IN_EVT | No | Yes ^(1.) | Yes | No |
| T_ANA_OUT_GEN | No | No | No | No |
| T_ANA_OUT_STD | No | No | No | No |
| T_ANA_OUT_STDX | No | No | Yes | No |
| T_ANA_IN_BMX | Yes | Yes | No | No |
| T_ANA_IN_T_BMX | Yes | Yes | No | No |
| T_ANA_OUT_BMX | Yes | Yes | No | No |
| T_ANA_IN_VE | No | No | No | No |
| T_ANA_IN_VWE | No | No | No | No |
| T_ANA_BI_VWE | No | No | No | No |
| T_ANA_BI_IN_VWE | No | No | No | No |
| Counting application | | | | |
| T_COUNT_ACQ | No | Yes ^(1.) | Yes | No |
| T_COUNT_HIGH_SPEED | No | Yes ^(1.) | Yes | No |
| T_COUNT_STD | No | Yes ^(1.) | Yes | No |
| T_SIGN_CPT_BMX | Yes | Yes | No | No |
| T_UNSIGN_CPT_BMX | Yes | Yes | No | No |
| T_CNT_105 | No | No | No | No |
| Electronic cam application | | | | |
| T_CCY_GROUP0 | No | No | No | No |
| T_CCY_GROUP1_2_3 | No | No | No | No |
| Axis control application | | | | |
| T_AXIS_AUTO | No | No | Yes | No |
| T_AXIS_STD | No | No | Yes | No |

| IODDT families | Modicon M340 | Modicon M580 | Premium | Quantum |
|----------------------------------|---------------------|---------------------|----------------|----------------|
| T_INTERPO_STD | No | No | Yes | No |
| T_STEPPER_STD | No | No | Yes | No |
| Sercos application | | | | |
| T_CSY_CMD | No | No | Yes | No |
| T_CSY_TRF | No | No | Yes | No |
| T_CSY_RING | No | No | Yes | No |
| T_CSY_IND | No | No | Yes | No |
| T_CSY_FOLLOW | No | No | Yes | No |
| T_CSY_COORD | No | No | Yes | No |
| T_CSY_CAM | No | No | Yes | No |
| Communication application | | | | |
| T_COM_STS_GEN | Yes | Yes | Yes | No |
| T_COM_UTW_M | No | No | Yes | No |
| T_COM_UTW_S | No | No | Yes | No |
| T_COM_MB | No | No | Yes | No |
| T_COM_CHAR | No | No | Yes | No |
| T_COM_FPW | No | No | Yes | No |
| T_COM_MBP | No | No | Yes | No |
| T_COM_JNET | No | No | Yes | No |
| T_COM_ASI | No | No | Yes | No |
| T_COM_ETY_1X0 | No | No | Yes | No |
| T_COM_ETY_210 | No | No | Yes | No |
| T_COM_IBS_128 | No | No | Yes | No |
| T_COM_IBS_242 | No | No | Yes | No |
| T_COM_PBY | No | No | Yes | No |
| T_COM_CPP100 | No | No | Yes | No |
| T_COM_ETYX103 | No | No | Yes | No |
| T_COM_ETHCOPRO | No | No | Yes | No |
| T_COM_MB_BMX | Yes | Yes | No | No |

| IODDT families | Modicon M340 | Modicon M580 | Premium | Quantum |
|---|---------------------|---------------------|----------------|----------------|
| T_COM_CHAR_BMX | Yes | Yes | No | No |
| T_COM_CO_BMX | Yes | Yes | No | No |
| T_COM_ETH_BMX | Yes | Yes | No | No |
| Adjustment application | | | | |
| T_PROC_PLOOP | No | No | Yes | No |
| T_PROC_3SING_LOOP | No | No | Yes | No |
| T_PROC_CASC_LOOP | No | No | Yes | No |
| T_PROC_SPP | No | No | Yes | No |
| T_PROC_CONST_LOOP | No | No | Yes | No |
| Weiging application | | | | |
| T_WEIGHING_ISPY101 | No | Yes ^(1.) | Yes | No |
| Common to all applications | | | | |
| T_GEN_MOD | No | No | No | No |
| 1. Premium module on extendable rack only | | | | |

Use of a DFB in a Ladder Language Program

Principle

In Ladder language, there are two possible ways of calling a DFB function block:

- via a textual call in an operation block in which the syntax and constraints on the parameters are identical to those of Structured Text language
- via a graphic call

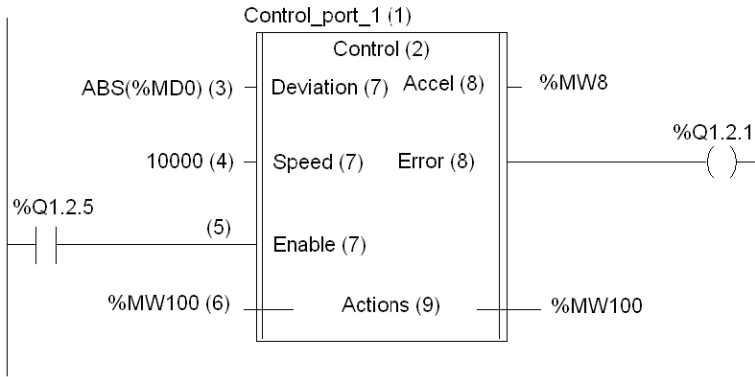
The inputs of the function blocks may be connected or assigned a value, an object or an expression. In any case, the type of external element (value, evaluation of the expression, etc.) must be identical to that of the input parameter.

A DFB block must have at least one connected Boolean input and an output (if necessary). For this you may use the EN input parameters and the ENO output parameter (see the description of these parameters below).

It is compulsory to connect or assign the ANY_ARRAY-type inputs, the generic data-type outputs (ANY_...) and the input/outputs of a DFB block.

Graphic Representation of a DFB Block

The following illustration shows a simple DFB programming example.



Elements of the DFB Block

The following table lists the different elements of the DFB block, labeled in the above illustration.

| Label | Element |
|-------|---|
| 1 | Name of the DFB (instance) |
| 2 | Name of the DFB type |
| 3 | Input assigned by an expression |
| 4 | Input assigned by a value |
| 5 | Connected input |
| 6 | Input assigned by an object (address or symbol) |
| 7 | Input parameters |
| 8 | Output parameters |
| 9 | Input/output parameters |

Use of EN\ENO Parameters

See EN and ENO Parameters, page 472

Use of a DFB in a Structured Text Language Program

Principle

In Structured Text, a user function block is called by a DFB call: name of the DFB instance followed by a list of arguments. Arguments are displayed in the list between brackets and separated by commas.

The DFB call can be of one of two types:

- a formal call, when arguments are assignments (parameter = value). In this case, the order in which the arguments are entered in the list is not important.
The EN input parameter and the ENO output parameter can be used to control the execution of the function block
- an informal call, when arguments are values (expression, object or an immediate value). In this case, the order in which the arguments are entered in the list must follow the order of the DFB input parameters, including for non-assigned inputs (the argument is an empty field)

It is not possible to use EN and ENO parameters.

DFB_Name (argument 1, argument 2, , argument n)

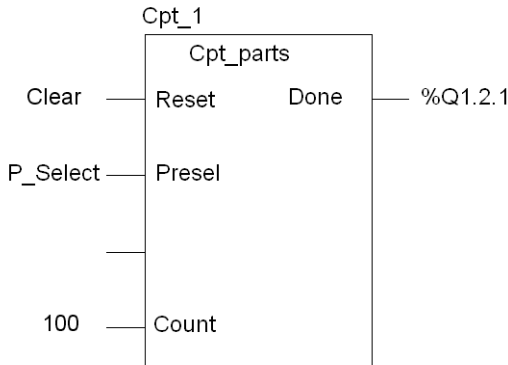
NOTE: The ANY_ARRAY-type inputs, generic data-type outputs (ANY_...) and input/outputs of a DFB must be assigned.

Use of EN\ENO Parameters

See EN and ENO Parameters, page 472

Example of a DFB

The following simple example explains the different DFB calls in Structured Text language. This is the instance `Cpt_1` of the `Cpt_parts`: type DFB.



Formal DFB Call

The formal DFB call `Cpt_1` is performed with the following syntax:

```
Cpt_1 (Reset:=Clear, Presel:=P_Select, Count:=100, Done=>%Q1.2.1);
```

Where the input parameters assigned by a value (expression, object or immediate value) are entered in the list of arguments, the syntax is:

```
Cpt_1 (Reset:=Clear, Presel:=P_Select, Count:=100);
```

...

```
%Q1.2.1:=Cpt_1.Done;
```

Elements of the Sequence

The following table lists the different elements of the program sequence, when a formal DFB call is made.

| Element | Meaning |
|-----------------------------------|-------------------------------|
| <code>Cpt_1</code> | Name of the DFB instance |
| <code>Reset, Presel, Count</code> | Input parameters |
| <code>:=</code> | Assignment symbol of an input |

| Element | Meaning |
|---------|--|
| Clear | Assignment object of an input (symbol) |
| 100 | Assignment value of an input |
| Done | Output parameter |
| => | Assignment symbol of an output |
| %Q1.2.1 | Assignment object of an output (address) |
| ; | End of sequence symbol |
| , | Argument separation symbol |

Informal DFB Call

The informal DFB call `Cpt_1` is performed with the following syntax:

```
Cpt_1 (Clear, %MD10, , 100);
...
%Q1.2.1:=Cpt_1.Done;
```

Elements of the Sequence

The following table lists the different elements of the program sequence, when a formal DFB call is made.

| Element | Meaning |
|---------------------------------|---|
| <code>Cpt_1</code> | Name of the DFB instance |
| <code>Clear, %MD10, ,100</code> | Assignment object or value of the inputs. Non-assigned inputs are represented by an empty field |
| ; | End of sequence symbol |
| , | Argument separation symbol |

Use of a DFB in an Instruction List Program

Principle

In Instruction List, a user function block is called by a `CAL` instruction, followed by the name of the DFB instance as an operand and a list of arguments (optional). Arguments are displayed in the list between brackets and separated by commas.

In Instruction List, there are three possible ways of calling a DFB:

- The instruction `CAL DFB_Name` is followed by a list of arguments that are assignments (parameter = value). In this case, the order in which the arguments are entered in the list is not important.

The EN input parameter can be used to control the execution of the function block.

- The instruction `CAL DFB_Name` is followed by a list of arguments that are values (expression, object or immediate value). In this case, the order in which the arguments are entered in the list must follow the order of the DFB input parameters, including for non-assigned inputs (the argument is an empty field).

It is not possible to use EN and ENO parameters.

- The instruction `CAL DFB_Name` is not followed by a list of arguments. In this case, this instruction must be preceded by the assignment of the input parameters, via a register: loading of the value (Load) then assignment to the input parameter (Store). The order of assignment of the parameters (LD/ST) is not important; however, you must assign all the required input parameters before executing the `CAL` command. It is not possible to use EN and ENO parameters.

```
CAL DFB_Name (argument 1, argument 2, ..., argument n)
```

or

```
LD Value 1
```

```
ST Parameter 1
```

```
...
```

```
LD Value n
```

```
ST Parameter n
```

```
CAL DFB_Name
```

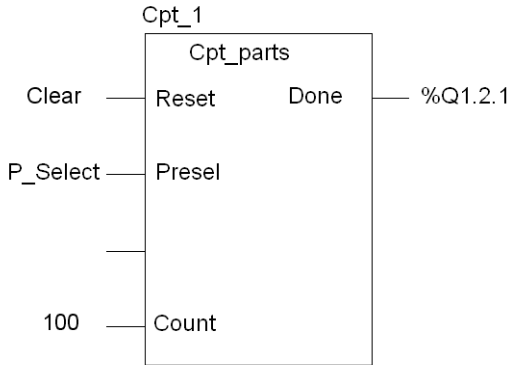
NOTE: The ANY_ARRAY-type inputs, generic data-type outputs (ANY_...) and input/outputs of a DFB must be assigned.

Use of EN\ENO Parameters

See EN and ENO Parameters, page 472.

Example of a DFB

The following example explains the different calls of a DFB in Instruction List. This is the instance `Cpt_1` of the `Cpt_parts`: type DFB



DFB Call when the Arguments Are Assignments

When the arguments are assignments, the DFB call `Cpt_1` is performed with the following syntax:

```
CAL Cpt_1 (Reset:=Clear, Presel:=%MD10, Count:=100, Done=>%Q1.2.1)
```

Where the input parameters assigned by a value (expression, object or immediate value) are entered in the list of arguments, the syntax is:

```
CAL Cpt_1 (Reset:=Clear, Presel:=%MD10, Count:=100)
```

...

```
LD Cpt_1.Done
```

```
ST %Q1.2.1
```

In order to make your application program more legible, you can enter a carriage return after the commas that separate the arguments. The sequence then takes the following syntax:

```
CAL Cpt_1 (
Reset:=Clear,
Presel:=%MD10,
Count:=100,
```



```
Done=>%Q1.2.1)
```

Elements of the DFB Call Program

The following table lists the different elements of the DFB call program.

| Element | Meaning |
|----------------------|--|
| CAL | DFB call instruction |
| Cpt_1 | Name of the DFB instance |
| Reset, Presel, Count | Input parameters |
| := | Assignment symbol of an input |
| Clear, %MD10, 100 | Assignment object or value of the inputs |
| Done | Output parameter |
| => | Assignment symbol of an output |
| %Q1.2.1 | Assignment object of an output |
| , | Argument separation symbol |

DFB Call when the Arguments Are Values

When the arguments are values, the DFB call `Cpt_1` is performed with the following syntax:

```
CAL Cpt_1 (Clear, %MD10, , 100)
```

```
...
```

```
LD Cpt_1.Done
```

```
ST %Q1.2.1
```

Elements of the DFB Call Program

The following table lists the different elements of the DFB call program.

| Element | Meaning |
|-------------------|--|
| CAL | DFB call instruction |
| Cpt_1 | Name of the DFB instance |
| Clear, %MD10, 100 | Assignment object or value of the inputs |
| , | Argument separation symbol |

DFB Call with no Argument

When there is no argument, the DFB call `Cpt_1` is performed with the following syntax:

```
LD Clear
ST Cpt_1.Reset
LD %MD10
ST Cpt_1.Presel
LD 100
ST Cpt_1.Count
CAL Cpt_1(
...
LD Cpt_1.Done
ST %Q1.2.1
```

Elements of the DFB Call Program

The following table lists the different elements of the DFB call program.

| Element | Meaning |
|----------------|---|
| LD Clear | Load instruction to load the <code>Clear</code> value into a register |
| ST Cpt_1.Reset | Assign instruction to assign the contents of the register to the input parameter <code>Cpt_1.Reset</code> |
| CAL Cpt_1(| Call instruction for the DFB <code>Cpt_1</code> |

Use of a DFB in a Program in Function Block Diagram Language

Principle

In FBD (Function Block Diagram) language, the user function blocks are represented in the same way as in Ladder language and are called graphically.

The inputs of the user function blocks may be connected or assigned a value, an immediate object or an expression. In any case, the type of external element must be identical to that of the input parameter.

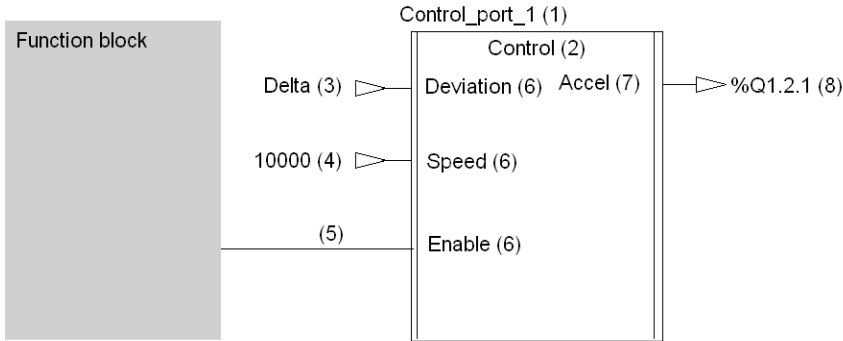
Only one object can be assigned (link to another block with the same variable) to an input of the DFB. However, several objects may be connected to a single output.

A DFB block must have at least one connected Boolean input and an output (if necessary). For this, you can use an EN input parameter and an ENO output parameter.

It is compulsory to connect or assign the ANY_ARRAY-type inputs, the generic data-type outputs (ANY_...) and the input/outputs of a DFB block.

Graphic Representation of a DFB Block

The following illustration shows a simple DFB programming example.



Elements of the DFB Block

The following table lists the different elements of the DFB block, labeled in the above illustration.

| Label | Element |
|-------|--------------------------------------|
| 1 | Name of the DFB (instance) |
| 2 | Name of the DFB type |
| 3 | Input assigned by an object (symbol) |
| 4 | Input assigned by a value |
| 5 | Connected input |
| 6 | Input parameters |

| Label | Element |
|-------|---------------------------------------|
| 7 | Output parameter |
| 8 | Input assigned by an object (address) |

Use of EN\ENO Parameters

See EN and ENO Parameters, page 472.

User Diagnostics DFB

What's in This Chapter

Presentation of User Diagnostic DFBs 501

Subject of this Chapter

This chapter describes how to create and use User Diagnostic (see EcoStruxure™ Control Expert, Diagnostics, Block Library) Function Blocks on Premium, Atrium and Quantum PLCs.

Presentation of User Diagnostic DFBs

General

The Control Expert application is used to create your own diagnostic DFBs (see EcoStruxure™ Control Expert, Operating Modes).

These diagnostic DFBs are standard DFBs that you will have configured beforehand with the **Diagnostic property** and in which you will have used the following two functions:

- REGDFB (see EcoStruxure™ Control Expert, Diagnostics, Block Library) to save the alarm date
- DEREK (see EcoStruxure™ Control Expert, Diagnostics, Block Library) to de-register the alarm

NOTE: It is strongly recommended to only program a diagnostic DFB instance once within the application.

These DFBs enable you to monitor your process. They will automatically report the information you will have chosen in the Viewer. You can thus monitor changes in state or variations in your process.

Advantages

The main advantages inherent in this service are as follows:

- The diagnostic is integrated in the project, and can thus be conceived during development and therefore better meets the user's requirements.
- The error dating and recording system is done at the source (in the PLC), which means the information exactly represents the state of the process.

- You can connect a number of Viewers (Control Expert, Magelis, Factory Cast) which will transcribe the exact state of the process to the user. Each Viewer is independent, and any action performed on one (for example, an acknowledgement) is automatically viewed on the others.

Implicit Type Conversion in Control Expert

What’s in This Chapter

Control Expert Implicit Type Conversion 503
 Control Expert Differences from IEC Recommendations 504

At a Glance

This chapter explains implicit type conversion in Control Expert.

Control Expert Implicit Type Conversion

Introduction

Control Expert provides a set of optional implicit type conversion. By checking the option **Enable implicit type conversion** in Project Settings (see EcoStruxure™ Control Expert, Operating Modes) the types conversions are implicitly done and you do not need to use most of the explicit type to type functions you used before.

Implicit Type Conversion Rules

After an implicit conversion, system bit %S18 (see EcoStruxure™ Control Expert, System Bits and Words, Reference Manual) is set to one to indicate a possible side-effect:

- loss of accuracy
- range mismatches
- unexpected implementation-dependent behavior

The formal test of the value of system bit %S18 is the responsibility of the programmer, the application must manage the behavior of its operative part.

NOTICE

UNINTENDED EQUIPMENT OPERATION

Check the system bit %S18 (via the application) after an implicit conversion.

Failure to follow these instructions can result in equipment damage.

The implicit type conversion rules:

Control Expert Differences

Control Expert has these exceptions to the IEC Recommendations:

1. If the result variable data type of an assignment is bigger than the result expression type, the parameters of the result expression are converted to the output parameter type to avoid an expression overflow.

Example:

```
i_DINT := INT1 + INT2;
```

Equivalent using explicit type conversion:

```
e_DINT := INT_TO_DINT(INT1) + INT_TO_DINT(INT2);
```

2. Control Expert uses implicit type conversion for generic functions, the data type of the result variable has an influence on the data type of the result expression (generic function).

Example:

```
i_DINT := ADD (IN1 := INT1, IN2 := INT2);
```

Equivalent using explicit type conversion:

```
e_DINT := ADD (IN1 := INT_TO_DINT(INT1), IN2 := INT_TO_DINT(INT2));
```

Generic output parameters of function blocks have no influence on the data type of the result expression.

Type conversions of non-matching input parameters are executed before the FFB-body is called and type conversions of output parameters are executed after the call. Implicit type conversions are, in contrast to explicit type conversions, only executed when the FFB-body is called.

Example:

```
SAH_0 (IN := BYTE1, CLK := BOOL1, PV := WORD1, OUT => i_DINT);
```

The next 3 lines are needed to get an equivalent result, using explicit type conversion:

```
word_tmp := DINT_TO_WORD(e_DINT);
```

```
SAH_0 (IN := BYTE_TO_WORD(BYTE1), CLK := BOOL1, PV := WORD1, OUT => _word_tmp);
```

```
e_DINT := WORD_TO_DINT(word_tmp);
```

The implicit type conversion rules are only applicable to typed constants. Control Expert treats untyped constants (literal values) initially as DINT constants.

Examples:

```
i_INT := 5 / 6 * 5.52;
```

Equivalent using explicit type conversion:

```
e_INT := REAL_TO_INT(DINT_TO_REAL(5) / DINT_TO_REAL(6) * 5.52);
```

```
i_BOOL := (65535 < INT1) = (BYTE1 = 255);
```

Equivalent using explicit type conversion:

```
e_BOOL := (65535 < INT_TO_DINT(INT1)) = (BYTE_TO_DINT(BYTE1) = 255);
```

Control Expert supports implicit type conversion inside expressions.

Examples:

```
i_INT := BYTE1 = DINT1;
```

Equivalent using explicit type conversion:

```
e_INT := BOOL_TO_INT(BYTE_TO_DINT(BYTE1) = DINT1);
```

```
i_WORD := BYTE1 = (REAL1 > DINT1);
```

Equivalent using explicit type conversion:

```
e_WORD := BOOL_TO_WORD(BYTE1 = BOOL_TO_BYTE((REAL1 > DINT_TO_REAL(DINT1))));
```

```
i_REAL := WORD1 OR BYTE1 AND (100000 + 5);
```

Equivalent using explicit type conversion:

```
e_REAL := DINT_TO_REAL(WORD_TO_DINT(WORD1) OR (BYTE_TO_DINT(BYTE1) AND (100000 + 5)));
```

Appendices

What's in This Part

IEC Compliance..... 508

At a Glance

The appendix contains additional information.

IEC Compliance

What’s in This Chapter

| | |
|---|-----|
| General Information regarding IEC 61131-3 | 508 |
| IEC Compliance Tables | 509 |
| Extensions of IEC 61131-3 | 529 |
| Textual language syntax | 531 |

Overview

This chapter contains the compliance tables required by IEC 61131-3.

General Information regarding IEC 61131-3

Overview

This section contains general information regarding IEC 61131-3 and its compliance requirements.

General information about IEC 61131-3 Compliance

At a Glance

The IEC 61131-3 Standard (cf. its subclause 1.4) specifies the syntax and semantics of a unified suite of programming languages for programmable controllers. These consist of two textual languages, IL (Instruction List) and ST (Structured Text), and two graphical languages, LD (Ladder Diagram) and FBD (Function Block Diagram).

Additionally, Sequential Function Chart (SFC) language elements are defined for structuring the internal organization of programmable controller programs and function blocks. Also, configuration elements are defined which support the installation of programmable controller programs into programmable controller systems.

NOTE: Control Expert uses the English acronyms for the programming languages.

Further more, features are defined which facilitate communication among programmable controllers and other components of automated systems.

Control Expert compliance to IEC 61131-3

The current version of the Control Expert Programming System supports a compliant subset of the language elements defined in the Standard.

Compliance in this context means the following:

- The Standard allows an implementer of an IEC Programming System to choose or to drop specific language features or even complete languages out of the Feature Tables which form an inherent part of the specifications; a system claiming compliance to the Standard just has to implement the chosen features according to the specifications given in the Standard.
- Further on, the Standard allows an implementer to use the defined programming language elements in an interactive programming environment. Since the Standard explicitly states that the specification of such environments is beyond its scope, the implementer has certain degrees of freedom in providing optimized presentation and handling procedures for specific language elements to the benefit of the user.
- Control Expert uses these degrees of freedom e.g. by introducing the notion of "Project" for the combined handling of the IEC language elements "Configuration" and "Resource". It also uses them e.g. in the mechanisms provided for handling variable declarations or function block instantiations.

IEC standards tables

The supported features and other implementation specific information is given in the following compliance statement and the subsequent tables as required by the Standard.

IEC Compliance Tables

Overview

This system complies with the requirements of IEC 61131-3 for the language and feature listed in the following tables.

Common elements

Common elements

IEC compliance table for common elements:

| Table No. | Feature No. | Description of Feature |
|-----------|-------------|------------------------|
| 1 | 2 | Lower case characters |

| Table No. | Feature No. | Description of Feature |
|-----------|-------------|--|
| | 3a | Number sign (#) |
| | 4a | Dollar sign (\$) |
| | 5a | Vertical bar () |
| 2 | 1 | Upper case and numbers |
| | 2 | Upper and lower case, numbers, embedded underlines |
| | 3 | Upper and lower case , numbers, leading or embedded underlines |
| 3 | 1 | Comments |
| 3a | 1 | Pragmas |
| 4 | 1 | Integer literals |
| | 2 | Real literals |
| | 3 | Real literals with exponents |
| | 4 | Base 2 literals |
| | 5 | Base 8 literals |
| | 6 | Base 16 literals |
| | 7 | Boolean zero and one |
| | 8 | Boolean FALSE and TRUE |
| | 9 | Typed literals |
| 5 | 1 | Single-byte character strings |
| | 3 | Single-byte typed string literals |
| 6 | 2 | Dollar sign |
| | 3 | Single quote |
| | 4 | Line feed |
| | 5 | New line |
| | 6 | Form feed (page) |
| | 7 | Carriage return |
| | 8 | Tab |
| | 9 | Double quote |
| 7 | 1a | Duration literals without underlines: short prefix |
| | 1b | long prefix |
| | 2a | Duration literals with underlines: short prefix |

| Table No. | Feature No. | Description of Feature |
|-----------|-------------|---|
| | 2b | long prefix |
| 8 | 1 | Date literals (long prefix) |
| | 2 | Date literals (short prefix) |
| | 3 | Time of day literals (long prefix) |
| | 4 | Time of day literals (short prefix) |
| | 5 | Date and time literals (long prefix) |
| | 5 | Date and time literals (short prefix) |
| 10 | 1 | Data type <code>BOOL</code> |
| | 3 | Data type <code>INT</code> |
| | 4 | Data type <code>DINT</code> |
| | 7 | Data type <code>UINT</code> |
| | 8 | Data type <code>UDINT</code> |
| | 10 | Data type <code>REAL</code> |
| | 12 | Data type <code>TIME</code> |
| | 13 | Data type <code>DATE</code> |
| | 14 | Data type <code>TIME_OF_DAY</code> or <code>TOD</code> |
| | 15 | Data type <code>DATE_AND_TIME</code> or <code>DT</code> |
| | 16 | Data type <code>STRING</code> |
| | 17 | Data type <code>BYTE</code> |
| | 18 | Data type <code>WORD</code> |
| | 19 | Data type <code>DWORD</code> |
| 12 | 4 | Array data types |
| | 5 | Structured data types |
| 14 | 4 | Initialization of array data types |
| | 6 | Initialization of derived structured data types |
| 15 | 1 | Input location |
| | 2 | Output location |
| | 3 | Memory location |
| | 4 | Single bit size (X Prefix) |

| Table No. | Feature No. | Description of Feature |
|-----------|-------------|--|
| | 5 | Single bit size (No Prefix) |
| | 7 | Word (16 bits) size |
| | 8 | Double word (32 bits) size |
| | 9 | Long (quad) word (64 bits) size |
| 17 | 3 | Declaration of locations of symbolic variables (Note 5, page 518) |
| | 4 | Array location assignment (Note 5, page 518) |
| | 5 | Automatic memory allocation of symbolic variables |
| | 6 | Array declaration (Note 11, page 520) |
| | 7 | Retentive array declaration (Note 11, page 520) |
| | 8 | Declaration for structured variables |
| 18 | 1 | Initialization of directly represented variables (Note 11, page 520) |
| | 3 | Location and initial value assignment to symbolic variables |
| | 4 | Array location assignment and initialization |
| | 5 | Initialization of symbolic variables |
| | 6 | Array initialization (Note 11, page 520) |
| | 7 | Retentive array declaration and initialization (Note 11, page 520) |
| | 8 | Initialization of structured variables |
| | 9 | Initialization of constants |
| | 10 | Initialization of function block instances |
| 19 | 1 | Negated input |
| | 2 | Negated output |
| 19a | 1 | formal function / function block call |
| | 2 | non-formal function / function block call |
| 20 | 1 | Use of EN and ENO shown in LD |
| | 2 | Usage without EN and ENO shown in FBD |
| 20a | 1 | In-out variable declaration (textual) |
| | 2 | In-out variable declaration (graphical) |
| | 3 | Graphical connection of in-out variable to different variables (graphical) |
| 21 | 1 | Overloaded functions |
| | 2 | Typed functions |

| Table No. | Feature No. | Description of Feature |
|-----------|-------------|--------------------------------|
| 22 | 1 | *_TO_* (Note 1, page 517) |
| | 2 | TRUNC (Note 2, page 518) |
| | 3 | *_BCD_TO_* (Note 3, page 518) |
| | 4 | **_TO_BCD_* (Note 3, page 518) |
| 23 | 1 | ABS function |
| | 2 | SQRT function |
| | 3 | LN function |
| | 4 | LOG function |
| | 5 | EXP function |
| | 6 | SIN function |
| | 7 | COS function |
| | 8 | TAN function |
| | 9 | ASIN function |
| | 10 | ACOS function |
| | 11 | ATAN function |
| 24 | 12 | ADD function |
| | 13 | MUL function |
| | 14 | SUB function |
| | 15 | DIV function |
| | 16 | MOD function |
| | 17 | EXPT function |
| | 18 | MOVE function |
| 25 | 1 | SHL function |
| | 2 | SHR function |
| | 3 | ROR function |
| | 4 | ROL function |
| 26 | 5 | AND function |
| | 6 | OR function |
| | 7 | XOR function |
| | 8 | NOT function |
| 27 | 1 | SEL function |
| | 2a | MAX function |

| Table No. | Feature No. | Description of Feature |
|-----------|-------------|-------------------------------------|
| | 2b | MIN function |
| | 3 | LIMIT function |
| | 4 | MUX function |
| 28 | 5 | GT function |
| | 6 | GE function |
| | 7 | EQ function |
| | 8 | LE function |
| | 9 | LT function |
| | 10 | NE function |
| 29 | 1 | LEN function (Note 4, page 518) |
| | 2 | LEFT function (Note 4, page 518) |
| | 3 | RIGHT function (Note 4, page 518) |
| | 4 | MID function (Note 4, page 518) |
| | 6 | INSERT function (Note 4, page 518) |
| | 7 | DELETE function (Note 4, page 518) |
| | 8 | REPLACE function (Note 4, page 518) |
| | 9 | FIND function (Note 4, page 518) |
| | 30 | 1a |
| 1b | | ADD_TIME function |
| 2b | | ADD_TOD_TIME function |
| 3b | | ADD_DT_TIME function |
| 4a | | SUB function (Note 6, page 520) |
| 4b | | SUB_TIME function |
| 5b | | SUB_DATE_DATE function |
| 6b | | SUB_TOD_TIME function |
| 7b | | SUB_TOD_TOD function |
| 8b | | SUB_DT_TIME function |
| 9b | | SUB_DT_DT function |
| 10a | | MUL function (Note 6, page 520) |
| 10b | | MULTIME function |

| Table No. | Feature No. | Description of Feature |
|-----------|-------------|---|
| | 11a | DIV function function (Note 6, page 520) |
| | 11b | DIVTIME function |
| 33 | 1a | RETAIN qualifier for internal variables (Note 11, page 520) |
| | 2a | RETAIN qualifier for output variables (Note 11, page 520) |
| | 2b | RETAIN qualifier for input variables (Note 11, page 520) |
| | 3a | RETAIN qualifier for internal function blocks (Note 11, page 520) |
| | 4a | VAR_IN_OUT declaration (textual) |
| | 4b | VAR_IN_OUT declaration and usage (graphical) |
| | 4c | VAR_IN_OUT declaration with assignment to different variables (graphical) |
| 34 | 1 | Bistable Function Block (set dominant) |
| | 2 | Bistable Function Block (reset dominant) |
| 35 | 1 | Rising edge detector |
| | 2 | Falling edge detector |
| 36 | 1a | CTU (Up-counter) function block |
| | 1b | CTU_DINT function block |
| | 1d | CTU_UDINT function block |
| | 2a | CTD (Down-counter) function block |
| | 2b | CTD_DINT function block |
| | 2d | CTD_UDINT function block |
| | 3a | CTUD (Up-down-counter) function block |
| | 3b | CTUD_DINT function block |
| | 3d | CTUD_UDINT function block |
| 37 | 1 | TP (Pulse) function block |
| | 2a | TON (On delay) function block |
| | 3a | TOF (Off delay) function block |
| 39 | 19 | Use of directly represented variables |
| 40 | 1 | Step and initial step - Graphical form with directed links |
| | 3a | Step flag – General form |
| | 4 | Step elapsed time– General form |

| Table No. | Feature No. | Description of Feature |
|-----------|-------------|--|
| 41 | 7 | Use of transition name |
| | 7a | Transition condition linked through transition name using LD language |
| | 7b | Transition condition linked through transition name using FBD language |
| | 7c | Transition condition linked through transition name using IL language |
| | 7d | Transition condition linked through transition name using ST language |
| 42 | 1 | Any Boolean variable declared in a VAR or VAR_OUTPUT block, or their graphical equivalents, can be an action |
| | 2l | Graphical declaration of action in LD language |
| | 2f | Graphical declaration of action in FBD language |
| | 3s | Textual declaration of action in ST language |
| | 3i | Textual declaration of action in IL language |
| 43 | 1 | Action block physically or logically adjacent to the step (Note 7, page 520) |
| | 2 | Concatenated action blocks physically or logically adjacent to the step (Note 8, page 520) |
| 44 | 1 | Action qualifier in action block supported |
| | 2 | Action name in action block supported |
| 45 | 1 | None - no qualifier |
| | 2 | Qualifier N |
| | 3 | Qualifier R |
| | 4 | Qualifier S |
| | 5 | Qualifier L |
| | 6 | Qualifier D |
| | 7 | Qualifier P |
| | 9 | Qualifier DS |
| | 11 | Qualifier P1 |
| | 12 | Qualifier P0 |
| 45a | 2 | Action control without "final scan" |
| 46 | 1 | Single sequence |
| | 2a | Divergence of sequence selection: left-to-right priority of transition evaluations |
| | 3 | Convergence of sequence selection |
| | 4 | Simultaneous sequences - divergence and convergence |

| Table No. | Feature No. | Description of Feature |
|-----------|-------------|--|
| | 5a | Sequence skip: left-to-right priority of transition evaluations |
| | 6a | Sequence loop: left-to-right priority of transition evaluations |
| 49 | 1 | CONFIGURATION . . . END_CONFIGURATION construction (Note 12, page 520) |
| | 5a | Periodic TASK construction |
| | 5b | Non-periodic TASK construction |
| | 6a | WITH construction for PROGRAM to TASK association (Note 9, page 520) |
| | 6c | PROGRAM declaration with no TASK association (Note 10, page 520) |
| 50 | 5a | Non-preemptive scheduling (Note 13, page 521) |
| | 5b | Preemptive scheduling (Note 14, page 521) |

Note 1

List of type conversion functions:

- `BOOL_TO_BYTE`, `BOOL_TO_DINT`, `BOOL_TO_INT`, `BOOL_TO_REAL`, `BOOL_TO_TIME`, `BOOL_TO_UDINT`, `BOOL_TO_UINT`, `BOOL_TO_WORD`, `BOOL_TO_DWORD`
- `BYTE_TO_BOOL`, `BYTE_TO_DINT`, `BYTE_TO_INT`, `BYTE_TO_REAL`, `BYTE_TO_TIME`, `BYTE_TO_UDINT`, `BYTE_TO_UINT`, `BYTE_TO_WORD`, `BYTE_TO_DWORD`, `BYTE_TO_BIT`
- `DINT_TO_BOOL`, `DINT_TO_BYTE`, `DINT_TO_INT`, `DINT_TO_REAL`, `DINT_TO_TIME`, `DINT_TO_UDINT`, `DINT_TO_UINT`, `DINT_TO_WORD`, `DINT_TO_DWORD`, `DINT_TO_DBCD`, `DINT_TO_STRING`
- `INT_TO_BOOL`, `INT_TO_BYTE`, `INT_TO_DINT`, `INT_TO_REAL`, `INT_TO_TIME`, `INT_TO_UDINT`, `INT_TO_UINT`, `INT_TO_WORD`, `INT_TO_BCD`, `INT_TO_DBCD`, `INT_TO_DWORD`, `INT_TO_STRING`
- `REAL_TO_BOOL`, `REAL_TO_BYTE`, `REAL_TO_DINT`, `REAL_TO_INT`, `REAL_TO_TIME`, `REAL_TO_UDINT`, `REAL_TO_UINT`, `REAL_TO_WORD`, `REAL_TO_DWORD`, `REAL_TO_STRING`
- `TIME_TO_BOOL`, `TIME_TO_BYTE`, `TIME_TO_DINT`, `TIME_TO_INT`, `TIME_TO_REAL`, `TIME_TO_UDINT`, `TIME_TO_UINT`, `TIME_TO_WORD`, `TIME_TO_DWORD`, `TIME_TO_STRING`
- `UDINT_TO_BOOL`, `UDINT_TO_BYTE`, `UDINT_TO_DINT`, `UDINT_TO_INT`, `UDINT_TO_REAL`, `UDINT_TO_TIME`, `UDINT_TO_UINT`, `UDINT_TO_WORD`, `UDINT_TO_DWORD`
- `UINT_TO_BOOL`, `UINT_TO_BYTE`, `UINT_TO_DINT`, `UINT_TO_INT`, `UINT_TO_REAL`, `UINT_TO_TIME`, `UINT_TO_UDINT`, `UINT_TO_WORD`, `UINT_TO_DWORD`,

- WORD_TO_BOOL, WORD_TO_BYTE, WORD_TO_DINT, WORD_TO_INT, WORD_TO_REAL, WORD_TO_TIME, WORD_TO_UDINT, WORD_TO_UINT, WORD_TO_BIT, WORD_TO_DWORD
- DWORD_TO_BOOL, DWORD_TO_BYTE, DWORD_TO_DINT, DWORD_TO_INT, DWORD_TO_REAL, DWORD_TO_TIME, DWORD_TO_UDINT, DWORD_TO_UINT, DWORD_TO_BIT,

The effects of each conversion are described in the help text supplied with the Base Library.

Note 2

List of types for truncate function:

- REAL_TRUNC_DINT, REAL_TRUNC_INT, REAL_TRUNC_UDINT, REAL_TRUNC_UINT

The effects of each conversion are described in the help text supplied with the Base Library.

Note 3

List of types for BCD conversion function:

- BCD_TO_INT, DBCD_TO_INT, DBCD_TO_DINT

List of types for BCD conversion function:

- INT_TO_BCD, INT_TO_DBCD, DINT_TO_DBCD

The effects of each conversion are described in the help text supplied with the Base Library.

Note 4

List of types for String functions:

- LEN_INT, LEFT_INT, RIGHT_INT, MID_INT, INSERT_INT, DELETE_INT, REPLACE_INT, FIND_INT

Note 5

A variable can be mapped to a directly represented variable if they strictly have the same type.

This means that a variable of type `INT` can only be mapped on a directly represented variable of type `INT`.

But there is one exception to this rule: for internal word (`%MW<i>`), Flat word (`%IW<i>`) and constant word (`%KW<i>`) memory variables any declared variable type is allowed.

Allowed mappings:

| | Syntax | Data type | Allowed variable types |
|----------------------|-----------------|------------------|--|
| Internal bit | %M<i> or %MX<i> | EBOOL | EBOOL ARRAY [..] OF EBOOL |
| Internal word | %MW<i> | INT | All types are allowed except: <ul style="list-style-type: none"> EBOOL ARRAY [..] OF EBOOL |
| Internal double word | %MD<i> | DINT | No mapping, because of overlapping between %MW<i> and %MD<i> and %MF<i>. |
| Internal real | %MF<i> | REAL | No mapping, because of overlapping between %MW<i> and %MD<i> and %MF<i>. |
| Constant word | %KW<i> | INT | All types are allowed except: <ul style="list-style-type: none"> EBOOL ARRAY [..] OF EBOOL |
| Constant double word | %KD<i> | DINT | No mapping, because of overlapping between %KW<i> and %KD<i> and %KF<i>. This kind of variables only exists on Premium PLCs. |
| Constant real | %KF<i> | REAL | No mapping, because of overlapping between %KW<i> and %KD<i> and %KF<i>. This kind of variables only exists on Premium PLCs. |
| System bit | %S<i> or %SX<i> | EBOOL | EBOOL |
| System word | %SW<i> | INT | INT |
| System double word | %SD<i> | DINT | DINT |
| Flat bit | %I<i> | EBOOL | EBOOL ARRAY [..] OF EBOOL This kind of variables only exists on Quantum PLCs. |
| Flat word | %IW<i> | INT | All types are allowed except: <ul style="list-style-type: none"> EBOOL ARRAY [..] OF EBOOL This kind of variables only exists on Quantum PLCs. |
| Common word | %NWi.j.k | INT | INT |

| | Syntax | Data type | Allowed variable types |
|-----------------------|-------------------|------------------|---|
| Topological variables | %I..., %Q..., ... | ... | Same Type (On some digital I/O modules it is allowed to map arrays of EBOOL on %IX<topo> and %QX<topo> objects.) |
| Extract bits | %MWi.j, ... | BOOL | BOOL |

Note 6

Only operator "+" (for ADD), "-" (for SUB), "*" (for MUL) or "/" (for DIV) in ST language.

Note 7

This feature is only presented in the "expanded view" of the chart.

Note 8

This feature is presented in the "expanded view" of the chart, but not as concatenated blocks, but as a scrollable list of action names with associated qualifiers inside one single block symbol.

Note 9

There is only a one-to-one mapping of program instance to task. The textual format is replaced by a property dialog.

Note 10

The textual format is replaced by a property dialog.

Note 11

All variables are retentive (RETAIN qualifier implicitly assumed in variable declarations).

Note 12

The textual format is replaced by the project browser representation.

Note 13

Using Mask-IT instruction, the user is able to get a non-preemptive behaviour. You will find `MASKEVT` (Global EVT masking) and `UNMASKEVT` (Global EVT unmasking) in the System functions of the libset.

Note 14

By default, the multi-task system is preemptive.

IL language elements

IL language elements

IEC compliance table for IL language elements:

| Table No. | Feature No. | Feature description |
|-----------|---------------------------------|---|
| 51b | 1 | Parenthesized expression beginning with explicit operator |
| 51b | 2 | Parenthesized expression (short form) |
| 52 | 1 | LD operator (with modifier "N") |
| | 2 | ST operator (with modifier "N") |
| | 3 | S, R operator |
| | 4 | AND operator (with modifiers "(", "N") |
| | 6 | OR operator (with modifiers "(", "N") |
| | 7 | XOR operator (with modifiers "(", "N") |
| | 7a | NOT operator |
| | 8 | ADD operator (with modifier "(") |
| | 9 | SUB operator (with modifier "(") |
| | 10 | MUL operator (with modifier "(") |
| | 11 | DIV operator (with modifier "(") |
| | 11a | MOD operator (with modifier "(") |
| | 12 | GT operator (with modifier "(") |
| | 13 | GE operator (with modifier "(") |
| | 14 | EQ operator (with modifier "(") |
| 15 | NE operator (with modifier "(") | |

| Table No. | Feature No. | Feature description |
|-----------|-------------|---|
| | 16 | LE operator (with modifier "(") |
| | 17 | LT operator (with modifier "(") |
| | 18 | JMP operator (with modifiers "C", "N") |
| | 19 | CAL operator (with modifiers "C", "N") |
| | 20 | RET operator (with modifiers "C", "N") (Note, page 522) |
| | 21 |) (evaluate deferred operation) |
| 53 | 1a | CAL of Function Block with non-formal argument list |
| | 1b | CAL of Function Block with formal argument list |
| | 2 | CAL of Function Block with load/store of arguments |
| | 4 | Function invocation with formal argument list |
| | 5 | Function invocation with non-formal argument list |

Note

In DFB only.

ST language elements

ST language elements

IEC compliance table for ST language elements:

| Table No. | Feature No. | Feature description |
|-----------|-------------|--|
| 55 | 1 | Parenthesization (expression) |
| | 2 | Function evaluation: functionName(listOfArguments) |
| | 3 | Exponentiation: ** |
| | 4 | Negation: - |
| | 5 | Complement: NOT |
| | 6 | Multiply: * |
| | 7 | Divide: / |
| | 8 | Modulo: MOD |
| | 9 | Add: + |

| Table No. | Feature No. | Feature description |
|-----------|-------------|---|
| | 10 | Subtract: - |
| | 11 | Comparison: <, >, <=, >= |
| | 12 | Equality: = |
| | 13 | Inequality: <> |
| | 14 | Boolean AND: & |
| | 15 | Boolean AND: AND |
| | 16 | Boolean Exclusive OR: XOR |
| | 17 | Boolean OR: OR |
| 56 | 1 | Assignment |
| | 2 | Function block invocation and function block output usage |
| | 3 | RETURN statement (Note, page 523) |
| | 4 | IF statement |
| | 5 | CASE statement |
| | 6 | FOR statement |
| | 7 | WHILE statement |
| | 8 | REPEAT statement |
| | 9 | EXIT statement |
| | 10 | Empty statement |

Note

In DFB only.

Common graphical elements

Common graphical elements

IEC compliance table for common graphical elements:

| Table No. | Feature No. | Feature description |
|-----------|-------------|--|
| 57 | 2 | Horizontal lines: Graphic or semigraphic |
| | 4 | Vertical lines: Graphic or semigraphic |
| | 6 | Horizontal/vertical connection: Graphic or semigraphic |

| Table No. | Feature No. | Feature description |
|-----------|-------------|---|
| | 8 | Line crossings without connection: Graphic or semigraphic |
| | 10 | Connected and non-connected corners: Graphic or semigraphic |
| | 12 | Blocks with connecting lines: Graphic or semi-graphic |
| 58 | 1 | Unconditional Jump: FBD Language |
| | 2 | Unconditional Jump: LD Language |
| | 3 | Conditional Jump: FBD Language |
| | 4 | Conditional Jump: LD Language |
| | 5 | Conditional Return: LD Language (Note, page 524) |
| | 6 | Conditional Return: FBD Language (Note, page 524) |
| | 7 | Unconditional Return from function or function block (Note, page 524) |
| | 8 | Unconditional Return: LD Language (Note, page 524) |

Note

In DFB only.

LD language elements

LD language elements

IEC compliance table for LD language elements:

| Table No. | Feature No. | Feature description |
|-----------|-------------|---|
| 59 | 1 | Left power rail |
| | 2 | Right power rail |
| 60 | 1 | Horizontal link |
| | 2 | Vertical link |
| 61 | 1 | Normally open contact (vertical bar) (Note, page 525) |
| | 3 | Normally closed contact (vertical bar) (Note, page 525) |
| | 5 | Positive transition-sensing contact (vertical bar) (Note, page 525) |
| | 7 | Negative transition-sensing contact (vertical bar) (Note, page 525) |
| 62 | 1 | Coil |

| Table No. | Feature No. | Feature description |
|-----------|-------------|----------------------------------|
| | 2 | Negated coil |
| | 3 | SET (latch) coil |
| | 4 | RESET (unlatch) coil |
| | 8 | Positive transition-sensing coil |
| | 9 | Negative transition-sensing coil |

Note

Only graphical representation.

Implementation-dependent parameters

Implementation-dependent parameters

IEC compliance table for implementation-dependent parameters:

| Parameters | Limitations/Behavior |
|--|---|
| Maximum length of identifiers | 32 characters |
| Maximum comment length | Within the Control Expert: 1024 characters for each editor object. Import: limited by XML constraints or UDBString usage in the persistent layer. |
| Syntax and semantics of pragmas | Unity Pro V1.0 only implements 1 pragma, used for legacy converter: <pre>{ ConvError (' error text'); }</pre> any other pragma construct is ignored (a warning message is given) NOTE: Unity Pro is the former name of Control Expert for version 13.1 or earlier. |
| Syntax and semantics for the use of the double-quote character when a particular implementation supports Feature #4 but not Feature #2 of Table 5. | (#2 of table 5 is supported) |
| Range of values and precision of representation for variables of type TIME, DATE, TIME_OF_DAY and DATE_AND_TIME | for TIME: t#0ms .. t#4294967295ms (=t#49D_17H_2M_47S_295MS) for DATE: D#1990-01-01 .. D#2099-12-31 |

| Parameters | Limitations/Behavior |
|---|---|
| | for TOD: TOD#00:00:00 .. TOD#23:59:59 |
| Precision of representation of seconds in types <code>TIME</code> , <code>TIME_OF_DAY</code> and <code>DATE_AND_TIME</code> | <p><code>TIME</code>: precision 1 ms</p> <p><code>TIME_OF_DAY</code>: precision 1 s</p> |
| Maximum number of enumerated values | Not applicable |
| Maximum number of array subscripts | 6 |
| Maximum array size | 64 kbytes |
| Maximum number of structure elements | no limit |
| Maximum structure size | no limit |
| Maximum range of subscript values | <code>DINT</code> range |
| Maximum number of levels of nested structures | 10 |
| Default maximum length of <code>STRING</code> and <code>WSTRING</code> variables | 16 characters |
| Maximum allowed length of <code>STRING</code> and <code>WSTRING</code> variables | 64 kbytes |
| Maximum number of hierarchical levels | Premium: physical mapping (5 levels) |
| Logical or physical mapping | Quantum: logical mapping (1 level) |
| Maximum number of inputs of extensible functions | <p>The number of all input parameters (including in-out parameters) is limited to 32. The number of all output parameters (including in-out parameters) is also limited to 32.</p> <p>Thus the limit for extensible input parameters is (32 - number of input parameters - number of in-out parameters).</p> <p>The limit for extensible output parameters is (32 - number of output parameters - number of in-out parameters).</p> |
| Effects of type conversions on accuracy | See online help. |
| Error conditions during type conversions | Error conditions are described in the online-help. Globally <code>%S18</code> is set for overflow errors detected. <code>ENO</code> is also set. The result is depending on the specific function. |
| Accuracy of numerical functions | INTEL floating point processing or emulation. |
| Effects of type conversions between time data types and other data types not defined in Table 30 | See online help. |
| Maximum number of function block specifications and instantiations | Only limited by the maximum size of a section. |
| Function block input variable assignment when <code>EN</code> is <code>FALSE</code> | No assignment |

| Parameters | Limitations/Behavior |
|---|--|
| Pvmin, Pymax of counters | <p>INT base counters:</p> <ul style="list-style-type: none"> Pvmin=-32768 (0x8000) Pymax=32767 (0x7FFF) <p>UINT base counters:</p> <ul style="list-style-type: none"> Pvmin=0 (0x0) Pymax=65535 (0xFFFF) <p>DINT base counters:</p> <ul style="list-style-type: none"> Pvmin=-2147483648 (0x80000000) Pymax=2147483647 (0x7FFFFFFF) <p>UDINT base counters:</p> <ul style="list-style-type: none"> Pvmin=0 (0x0) Pymax=4294967295 (0xFFFFFFFF) |
| Effect of a change in the value of a PT input during a timing operation | The new PT values are immediately taken at once into account, even during a running timing operation immediately works with the new values. |
| Program size limitations | Depends on controller type and memory |
| Precision of step elapsed time | 10 ms |
| Maximum number of steps per SFC | 1024 steps per SFC section |
| Maximum number of transitions per SFC and per step | <p>Limited by the available area for entering steps/transitions and by the maximum number of steps per SFC section (1024 Steps).</p> <p>32 transition per step. Limited by the available area for entering Alternative/Parallel branches, maximum is 32 rows.</p> |
| Maximum number of action blocks per step | 20 |
| Access to the functional equivalent of the Q or A outputs | not applicable |
| Transition clearing time | <p>Target dependent;</p> <p>always < 100 micro-seconds</p> |
| Maximum width of diverge/converge constructs | 32 |
| Contents of RESOURCE libraries | Not applicable |
| Effect of using READ_WRITE access to function block outputs | Not applicable |
| Maximum number of tasks | <p>Depends on controller type.</p> <p>Maximum on most powerful controller: 9 tasks</p> |
| Task interval resolution | 10 ms |
| Maximum length of expressions | Practically no limit |

| Parameters | Limitations/Behavior |
|--|--|
| Maximum length of statements | Practically no limit |
| Maximum number of CASE selections | Practically no limit |
| Value of control variable upon termination of FOR loop | Undefined |
| Restrictions on network topology | No restrictions |
| Evaluation order of feedback loops | The block connected to the feedback variable is executed first |

Error Conditions

Error Conditions

IEC standards table for error conditions:

| Error conditions | Treatment (see Note, page 529) |
|---|---|
| Nested comments | 2) error is reported during programming |
| Value of a variable exceeds the specified subrange | 4) error is reported during execution |
| Missing configuration of an incomplete address specification ("**" notation) | Not applicable |
| Attempt by a program organization unit to modify a variable which has been declared <code>CONSTANT</code> | 2) error is reported during programming |
| Improper use of directly represented or external variables in functions | Not applicable |
| A <code>VAR_IN_OUT</code> variable is not "properly mapped" | 2) error is reported during programming |
| Type conversion errors | 4) error is reported during execution |
| Numerical result exceeds range for data type | 4) error is reported during execution |
| Division by zero | 4) error is reported during execution |
| Mixed input data types to a selection function | 2) error is reported during programming |
| Result exceeds range for data type | 4) error is reported during execution |
| No value specified for an in-out variable | 2) error is reported during programming |
| Zero or more than one initial steps in SFC network | 3) error is reported during analyzing/loading/linking |
| User program attempts to modify step state or time | 2) error is reported during programming |
| Side effects in evaluation of transition condition | 3) error is reported during analyzing/loading/linking |
| Action control contention error | 3) error is reported during analyzing/loading/linking |

| Error conditions | Treatment (see Note, page 529) |
|--|---|
| Simultaneously true, non-prioritized transitions in a selection divergence | Not applicable |
| Unsafe or unreachable SFC | 3) error is reported during analyzing/loading/linking |
| Data type conflict in VAR_ACCESS | Not applicable |
| A task fails to be scheduled or to meet its execution deadline | 4) error is reported during execution |
| Numerical result exceeds range for data type | 4) error is reported during execution |
| Current result and operand not of same data type | 2) error is reported during programming |
| Division by zero | 4) error is reported during execution |
| Numerical result exceeds range for data type | 4) error is reported during execution |
| Invalid data type for operation | 4) error is reported during execution |
| Return from function without value assigned | Not applicable |
| Iteration fails to terminate | 4) error is reported during execution |
| Same identifier used as connector label and element name | Not applicable |
| Uninitialized feedback variable | 1) error is not reported |

Note

Identifications for the treatment of error conditions according to IEC 61131-3, subclause 1.5.1, d):

- 1) error is not reported
- 2) error is reported during programming
- 3) error is reported during analyzing/loading/linking
- 4) error is reported during execution

Extensions of IEC 61131-3

Overview

This section describes the extensions of IEC 61131-3

Extensions of IEC 61131-3, 2nd Edition

At a Glance

In addition to the standardized IEC features listed in the IEC Compliance Tables, page 509, the Control Expert programming environment inherited a number of features from the PL7 programming environment. These extensions are optionally provided; they can be checked or not in a corresponding options dialog. The dialog and the features are described in detail in a chapter of the online help titled *Data and Languages* (see EcoStruxure™ Control Expert, Operating Modes).

Not included in the options dialog is another extension, which is inherited both from the PL7 and the Concept programming environments: Control Expert provides the construct of the so-called Section in all programming languages, which allows to subdivide a Program Organization Unit (POU). This construct introduces the possibility to mix several languages (e.g. FBD sections, SFC sections) in a POU body, a feature which, if used for this purpose, constitutes an extension of the IEC syntax. A compliant POU body should contain a single section only. Sections or Program Units do not create a distinct name scope; the name scope for all language elements is the POU.

Purpose of Sections, or Program Units

Sections, or Program Units serve different purposes:

- Sections, or Program Units allow to subdivide large POU bodies according to functional aspects: the user has the possibility to subdivide his POU body into functionally meaningful parts. The list of sections represents a kind of functional table of contents of a large, otherwise unstructured POU body.
- Sections, or Program Units allow to subdivide large POU bodies according to graphical aspects: the user has the possibility to design substructures of a large POU body according to an intended graphical presentation. He can create small or large graphical sections according to his taste.
- The subdivision of large POU bodies allows quick online changes: in Control Expert, the Section, or the Program Unit serves as the unit for online change. If a POU body is modified during runtime at different locations, automatically all sections affected by the changes are downloaded on explicit request.
- Sections, or Program Units allow to rearrange the execution order of specific, labeled parts of a POU body: the section name serves as a label of that part of the body which is contained inside the section, and by ordering these labels the execution order of those parts is manageable.
- Sections, or Program Units allow to use different languages in parallel in the same POU: this feature is a major extension of the IEC syntax, which allows only one single IEC language to be used for a POU body. In a compliant body, SFC has to be used to manage different languages inside a body (each transition and action may be formulated in its own language).

Textual language syntax

Overview

This section describes the textual language syntax

Textual Language Syntax

Description

The Control Expert programming environment does not yet provide support for an import or export of text files complying with the full textual language syntax as specified in Annex B of IEC 61131-3, 2nd Edition.

However, the textual syntax of the IL and ST languages, as specified in Annex B.2 and B.3 of IEC 61131-3, 2nd Edition, including all directly and indirectly referenced productions out of Annex B.1, is supported in textual language sections.

Those syntax productions in Annex B of IEC 61131-3, 2nd Edition belonging to features which are not supported by Control Expert according to the compliance tables, page 509 are not implemented.

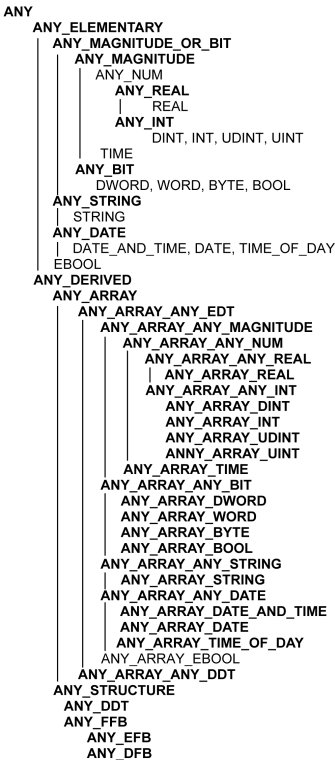
Glossary

A

ANY:

There is a hierarchy between the different types of data. In the DFB, it is sometimes possible to declare which variables can contain several types of values. Here, we use ANY_xxx types.

The following diagram shows the hierarchically-ordered structure:



ARRAY:

An `ARRAY` is a table of elements of the same type.

The syntax is as follows: `ARRAY [<terminals>] OF <Type>`

Example:

`ARRAY [1..2] OF BOOL` is a one-dimensional table made up of two `BOOL`-type elements.

`ARRAY [1..10, 1..20] OF INT` is a two-dimensional table made up of 10x20 `INT`-type elements.

Auxiliary tasks:

Optional periodic tasks used to process procedures that do not require fast processing: measurement, adjustment, diagnostic aid, etc.

B

BOOL:

`BOOL` is the abbreviation for Boolean type. This is the elementary data item in computing. A `BOOL` type variable has a value of either: 0 (`FALSE`) or 1 (`TRUE`).

A `BOOL` type word extract bit, for example: `%MW10.4`.

BYTE:

When 8 bits are put together, this is called a `BYTE`. A `BYTE` is either entered in binary, or in base 8.

The `BYTE` type is coded in an 8 bit format, which, in hexadecimal, ranges from `16#00` to `16#FF`

C

cyclic execution:

The master task is executed either cyclically or periodically. Cyclical execution consists of stringing cycles together one after the other with no waiting time between the cycles.

D**DATE:**

The `DATE` type coded in BCD in 32 bit format contains the following information:

- the year coded in a 16-bit field,
- the month coded in an 8-bit field,
- the day coded in an 8-bit field.

The `DATE` type is entered as follows: `D#<Year>-<Month>-<Day>`

This table shows the lower/upper limits in each field:

| Field | Limits | Comment |
|-------|-------------|---|
| Year | [1990,2099] | Year |
| Month | [01,12] | The left 0 is always displayed, but can be omitted at the time of entry |
| Day | [01,31] | For the months 01\03\05\07\08\10\12 |
| | [01,30] | For the months 04\06\09\11 |
| | [01,29] | For the month 02 (leap years) |
| | [01,28] | For the month 02 (non leap years) |

DDT:

DDT is the abbreviation for Derived Data Type.

A derived data type is a set of elements of the same type (`ARRAY`) or of various types (structure)

DFB:

DFB is the abbreviation for Derived Function Block.

DFB types are function blocks that can be programmed by the user ST, IL, LD or FBD.

By using DFB types in an application, it is possible to:

- simplify the design and input of the program,
- increase the legibility of the program,
- facilitate the debugging of the program,
- reduce the volume of the generated code.

DINT:

DINT is the abbreviation for Double Integer format (coded on 32 bits).

The lower and upper limits are as follows: $-(2 \text{ to the power of } 31)$ to $(2 \text{ to the power of } 31) - 1$.

Example:

`-2147483648, 2147483647, 16#FFFFFFFF.`

DT:

DT is the abbreviation for Date and Time.

The DT type coded in BCD in 64 bit format contains the following information:

- The year coded in a 16-bit field,
- the month coded in an 8-bit field,
- the day coded in an 8-bit field,
- the hour coded in a 8-bit field,
- the minutes coded in an 8-bit field,
- the seconds coded in an 8-bit field.

NOTE: The 8 least significant bits are unused.

The DT type is entered as follows:

`DT#<Year>-<Month>-<Day>-<Hour>:<Minutes>:<Seconds>`

This table shows the lower/upper limits in each field:

| Field | Limits | Comment |
|--------|-------------|---|
| Year | [1990,2099] | Year |
| Month | [01,12] | The left 0 is always displayed, but can be omitted at the time of entry |
| Day | [01,31] | For the months 01\03\05\07\08\10\12 |
| | [01,30] | For the months 04\06\09\11 |
| | [01,29] | For the month 02 (leap years) |
| | [01,28] | For the month 02 (non leap years) |
| Hour | [00,23] | The left 0 is always displayed, but can be omitted at the time of entry |
| Minute | [00,59] | The left 0 is always displayed, but can be omitted at the time of entry |
| Second | [00,59] | The left 0 is always displayed, but can be omitted at the time of entry |

DWORD:

DWORD is the abbreviation for Double Word.

The DWORD type is coded in 32 bit format.

This table shows the lower/upper limits of the bases which can be used:

| Base | Lower limit | Upper limit |
|-------------|-------------|------------------------------------|
| Hexadecimal | 16#0 | 16#FFFFFFFF |
| Octal | 8#0 | 8#3777777777 |
| Binary | 2#0 | 2#11111111111111111111111111111111 |

Representation examples:

| Data Content | Representation in One of the Bases |
|----------------------------------|------------------------------------|
| 00000000000010101101110011011110 | 16#ADCDE |
| 00000000000000100000000000000000 | 8#200000 |
| 00000000000010101011110011011110 | 2#10101011110011011110 |

E**EBOOL:**

EBOOL is the abbreviation for Extended Boolean type. A EBOOL type variable brings a value 0 (FALSE) or 1 (TRUE) but also rising or falling edges and forcing capabilities.

An EBOOL type variable takes up one byte of memory.

The byte split up into:

- one bit for the value,
- one bit for the history bit (each time the state's object changes, the value is copied inside the history bit),
- one bit for the forcing bit (equals to 0 if the object isn't forced, equal to 1 if the bit is forced).

The default type value of each bit is 0 (FALSE).

EDT:

EDT is the abbreviation for Elementary Data Type.

These types are as follows:

- BOOL,
- EBOOL,
- WORD,
- DWORD,
- INT,
- DINT,
- UINT,
- UDINT,
- REAL,
- DATE,
- TOD,
- DT.

EFB:

Is the abbreviation for Elementary Function Block.

This is a block which is used in a program, and which performs a predefined software function.

EFBs have internal statuses and parameters. Even where the inputs are identical, the output values may be different. For example, a counter has an output which indicates that the preselection value has been reached. This output is set to 1 when the current value is equal to the preselection value.

event processing:

Event processing 1 is a program section launched by an event. The instructions programmed in this section are executed when a software application event (Timer) or a hardware event (application specific module) is received by the processor.

Event processes take priority over other tasks, and are executed the moment the event is detected.

The event process EVT0 is of highest priority. All others have the same level of priority.

NOTE: For M340, IO events with the same priority level are stored in a FIFO and are treated in the order in which they are received.

All the timers have the same priority. When several timers end at the same time, the lowest timer number is processed first.

The system word %SW48 counts IO events and telegram processed.

NOTE: TELEGRAM is available only for PREMIUM (not on Quantum or M340)

F**FAST task:**

Task launched periodically (setting of the period in the PC configuration) used to carry out a part of the application having a superior level of priority to the Mast task (master).

I**instantiate:**

To instantiate an object is to allocate a memory space whose size depends on the type of object to be instantiated. When an object is instantiated, it exists and can be manipulated by the program.

I/O object:

An I/O object is an implicit or explicit language object for an expert function module or a I/O device on a fieldbus. They are of the following types: %Ch, %I, %IW, %ID, %IF, %Q, %QW, %QD, %QF, %KW, %KD, %KF, %MW, %MD, and %MF.

The objects' topological address depends on the module's position on the rack or the device's position on the bus.

For Premium/Atrium PLCs double-type instances of located data (%MD<i>, %KD<i>) or floating (%MF<i>, %KF<i>) should be located by an integer type (%MW<i>, %KW<i>). Only I/O objects make it possible to locate type instances (%MD<i>, %KD<i>, %QD, %ID, %MF<i>, %KF<i>, %QF, %IF) by using their topological address (for example %MD0.6.0.11, %MF0.6.0.31).

For Modicon M340 PLCs, double-type instances of located data (%MD<i>, %KD<i>) or floating (%MF<i>, %KF<i>) are not available.

INT:

INT is the abbreviation for single integer format (coded on 16 bits).

The lower and upper limits are as follows: -(2 to the power of 31) to (2 to the power of 31) - 1.

Example:

-32768, 32767, 2#1111110001001001, 16#9FA4.

IODDT:

IODDT is the abbreviation for Input/Output Derived Data Type.

The term IODDT designates a structured data type representing a module or a channel of a PLC module. Each application expert module possesses its own IODDTs.

K

keyword:

A keyword is a unique combination of characters used as a syntactical programming language element (See annex B definition of the IEC standard 61131-3. All the key words used in Control Expert and of this standard are listed in annex C of the IEC standard 61131-3. These keywords cannot be used as identifiers in your program (names of variables, sections, DFB types, etc.)).

M

MAST task:

Main program task.

It is obligatory and is used to carry out sequential processing of the PLC.

multiple token:

Operating mode of an SFC. In multitoken mode, the SFC may possess several active steps at the same time.

N

naming convention (identifier):

An identifier is a sequence of letters, numbers and underlines beginning with a letter or underline (e.g., name of a function block type, an instance, a variable or a section). If you select the **Extended** option in the **Tools > Project Settings... > Variables** dialog, letters from national character sets (e.g., ö, ü, é, ð) can be used. Underlines are significant in identifiers; e.g., A BCD and AB CD are interpreted as different identifiers. Ending underlines is invalid.

Identifiers cannot contain spaces. Not case sensitive; e.g., ABCD and abcd are interpreted as the same identifier.

According to IEC 61131-3 leading digits are not allowed in identifiers. Nevertheless, you can use them if you activate the check box **Allow leading digits** in the **Tools > Project Settings... > Variables** dialog.

According to IEC 61131-3 multiple leading underlines and consecutives underlines are not allowed in identifiers. Nevertheless, you can use them if you select the **Extended** option in the **Tools > Project Settings... > Variables > Character set** dialog.

Identifiers cannot be keywords.

P

periodic execution:

The master task is executed either cyclically or periodically. In periodic mode, you determine a specific time (period) in which the master task must be executed. If it is executed under this time, a waiting time is generated before the next cycle. If it is executed over this time, a control system indicates the overrun. If the overrun is too high, the PLC is stopped.

Program Unit:

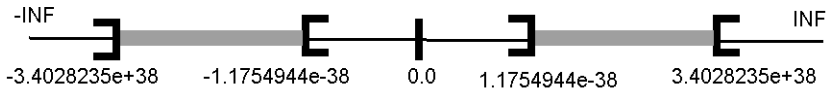
A Program Unit is a part of program with it's own set of local and public variables. Program Units allow easy duplication and clear organization of program with local and public variables. Program Units are compliant with Program Organization Units (POUs) program as defined in IEC1131-3 standard.

R

REAL:

Real type is a coded type in 32 bits.

The ranges of possible values are illustrated in gray in the following diagram:



When a calculation result is:

- between $-1.175494e-38$ and $1.175494e-38$ it is considered as a DEN,
- less than $-3.4028234e+38$, the symbol $-INF$ (for - infinite) is displayed,
- greater than $+3.4028234e+38$, the symbol INF (for +infinite) is displayed,
- undefined (square root of a negative number), the symbol NAN or NAN is displayed.

NOTE: The IEC 559 standard defines two classes of NAN: quiet NAN (QNaN) and signaling NaN (SNaN) QNaN is a NAN with the most significant fraction bit set and a SNaN is a NAN with the most significant fraction bit clear (Bit number 22). QNaNs are allowed to propagate through most arithmetic operations without signaling an exception. SNaN generally signal an invalid-operation exception whenever they appear as operands in arithmetic operations (See %SW17 and %S18).

NOTE: when an operand is a DEN (De-normalized number) the result is not significant.

S

section:

Program module belonging to a task which can be written in the language chosen by the programmer (FBD, LD, ST, IL, or SFC).

A task can be composed of several sections, the order of execution of the sections corresponding to the order in which they are created, and being modifiable.

SFC objects:

An SFC object is a data structure representing the status properties of an action or transition of a sequential chart.

single token:

Operating mode of an SFC chart for which only a single step can be active at any one time.

STRING:

A variable of the type `STRING` is an ASCII standard character string. A character string has a maximum length of 65534 characters.

structure:

View in the project navigator with represents the project structure.

subroutine:

Program module belonging to a task (MAST, FAST, AUX) which can be written in the language chosen by the programmer (FBD, LD, ST, or IL).

A subroutine may only be called by a section or by another subroutine belonging to the task in which it is declared.

T**TIME:**

The type `TIME` expresses a duration in milliseconds. Coded in 32 bits, this type makes it possible to obtain periods from 0 to $(2^{32}-1)$ milliseconds.

TOD:

TOD is the abbreviation for Time of Day.

The TOD type coded in BCD in 32 bit format contains the following information:

- the hour coded in a 8-bit field,
- the minutes coded in an 8-bit field,
- the seconds coded in an 8-bit field.

NOTE: The 8 least significant bits are unused.

The Time of Day type is entered as follows: **TOD#**<Hour>:<Minutes>:<Seconds>

This table shows the lower/upper limits in each field:

| Field | Limits | Comment |
|--------|---------|---|
| Hour | [00,23] | The left 0 is always displayed, but can be omitted at the time of entry |
| Minute | [00,59] | The left 0 is always displayed, but can be omitted at the time of entry |
| Second | [00,59] | The left 0 is always displayed, but can be omitted at the time of entry |

Example: TOD#23:59:45.

U

UDINT:

UDINT is the abbreviation for Unsigned Double Integer format (coded on 32 bits) unsigned. The lower and upper limits are as follows: 0 to (2 to the power of 32) - 1.

Example:

0, 4294967295, 2#11111111111111111111111111111111, 8#377777777777, 16#FFFFFFFF.

UINT:

UINT is the abbreviation for Unsigned integer format (coded on 16 bits). The lower and upper limits are as follows: 0 to (2 to the power of 16) - 1.

Example:

0, 65535, 2#1111111111111111, 8#177777, 16#FFFF.

W**WORD:**

The **WORD** type is coded in 16 bit format and is used to carry out processing on bit strings.

This table shows the lower/upper limits of the bases which can be used:

| Base | Lower Limit | Upper Limit |
|-------------|-------------|--------------------|
| Hexadecimal | 16#0 | 16#FFFF |
| Octal | 8#0 | 8#177777 |
| Binary | 2#0 | 2#1111111111111111 |

Representation examples

| Data Content | Representation in One of the Bases |
|------------------|------------------------------------|
| 0000000011010011 | 16#D3 |
| 1010101010101010 | 8#125252 |
| 0000000011010011 | 2#11010011 |

Index

| | |
|--|---------------|
| A | |
| ADD | |
| IL..... | 387 |
| addressing | |
| data instances | 233 |
| input/output | 233 |
| alignment | |
| DDT..... | 199 |
| AND | |
| IL..... | 385 |
| ST | 428 |
| ANY_ARRAY | 214 |
| ANY_BOOL..... | 168 |
| ARRAY..... | 192 |
| automatic start in RUN..... | 135 |
| B | |
| BOOL..... | 168 |
| BYTE | 190 |
| C | |
| CAL | 391 |
| CASE...OF...END_CASE | |
| ST | 435 |
| channel data structure | 203–204 |
| cold start | 135, 144 |
| comparison | |
| IL..... | 381 |
| LD | 295 |
| ST | 425 |
| compatibility | |
| data types | 218 |
| D | |
| D | |
| SFC..... | 335 |
| data instances | 226 |
| data type | |
| Reference | 222 |
| data types..... | 166 |
| DATE | 178 |
| DDT | 192 |
| alignment | 199 |
| derived data types (DDT)..... | 192, 195 |
| derived function block (DFB) | 463 |
| representation | 208, 466 |
| device DDT | |
| Instance name..... | 205 |
| Device DDT Instance | |
| name | 205 |
| Device derived data types (DDDT) | 192 |
| DFB | |
| representation | 466 |
| diagnostics DFB..... | 501 |
| DINT | 174 |
| DIV | |
| IL..... | 388 |
| DS | |
| SFC..... | 335 |
| DT | 180 |
| DWORD..... | 190 |
| E | |
| EBOOL | 168 |
| EDT | 166 |
| EFB | 208 |
| elementary data types (EDT)..... | 166 |
| elementary function block (EFB)..... | 208 |
| ELSE | 433 |
| ELSIF...THEN | 434 |
| EN/ENO | |
| FBD..... | 262 |
| IL..... | 400, 410, 417 |
| LD | 291 |
| ST | 447, 455, 460 |
| EQ | |
| IL..... | 389 |
| event | |
| timer | 111 |
| event processing..... | 102 |
| EXIT | 439 |
| F | |
| FBD | |

| | |
|---|----------|
| language..... | 253, 255 |
| structure..... | 253 |
| final scan..... | 335 |
| floating point..... | 182 |
| FOR...TO...BY...DO...END_FOR | |
| ST..... | 435 |
| forced bits..... | 168 |
| functions available for the different types | |
| of PLC..... | 80 |

G

| | |
|----------|-----|
| GDT..... | 214 |
| GE | |
| IL..... | 389 |
| GT | |
| IL..... | 389 |

H

| | |
|-----------|-----|
| HALT..... | 156 |
|-----------|-----|

I

| | |
|-------------------------------|--------------------|
| IEC Compliance..... | 508 |
| IF...THEN...END_IF | |
| ST..... | 432 |
| implicit type conversion..... | 503 |
| Implicit Type Conversion..... | 503 |
| IN_OUT | |
| FBD..... | 264 |
| IL..... | 411, 418 |
| LD..... | 293 |
| ST..... | 455, 461 |
| input/output | |
| addressing..... | 233 |
| instruction list (IL) | |
| language..... | 374, 395, 401, 412 |
| operators..... | 381 |
| structure..... | 374 |
| INT..... | 174 |

J

| | |
|----------|-----|
| JMP | |
| FBD..... | 266 |

| | |
|----------|---------|
| IL..... | 392–393 |
| LD..... | 295 |
| SFC..... | 345 |
| ST..... | 441 |

L

| | |
|----------------|----------|
| L | |
| SFC..... | 335 |
| labels | |
| FBD..... | 266 |
| IL..... | 393 |
| LD..... | 295 |
| ST..... | 441 |
| LD | |
| language..... | 278, 283 |
| structure..... | 278 |
| LD operators | |
| IL..... | 278 |
| LE | |
| IL..... | 390 |
| LT..... | 390 |

M

| | |
|------------------------|-----|
| memory structures..... | 119 |
| Modicon M340..... | 123 |
| MOD | |
| IL..... | 388 |
| ST..... | 426 |
| Modicon M340 | |
| memory structures..... | 123 |
| State RAM..... | 123 |
| MUL | |
| IL..... | 387 |

N

| | |
|--------------------------|-----|
| name | |
| device DDT Instance..... | 205 |
| Device DDT Instance..... | 205 |
| NE | |
| IL..... | 389 |
| NOT | |
| IL..... | 386 |

| | | | |
|----------------------------------|--------------------|---------------------------|--------------------|
| O | | SFC..... | 335 |
| operate..... | 295 | sections..... | 92 |
| OR | | SFC | |
| IL..... | 385 | language..... | 319, 332 |
| ST..... | 429 | structure..... | 319 |
| P | | SFCCHART_STATE..... | 322 |
| P | | SFCSTEP_STATE..... | 326 |
| SFC..... | 335 | SFCSTEP_TIMES..... | 326 |
| P0 | | State RAM | |
| SFC..... | 335 | Modicon M340..... | 123 |
| P1 | | state RAM of Modicon M340 | |
| SFC..... | 335 | RUN mode..... | 145 |
| private variables | | STOP mode..... | 145 |
| DFB..... | 475 | STRING..... | 187 |
| FBD..... | 261, 290, 403, 450 | structure..... | 192 |
| Program Unit..... | 90 | structured text (ST) | |
| public variables | | instructions..... | 429 |
| DFB..... | 475 | language..... | 420, 442, 448, 457 |
| FBD..... | 260 | operators..... | 425 |
| IL..... | 403 | structure..... | 420 |
| LD..... | 289 | SUB | |
| ST..... | 450 | IL..... | 387 |
| R | | subroutines..... | 92, 95 |
| R | | T | |
| IL..... | 384 | tasks..... | 85, 87 |
| LD..... | 282 | cyclic..... | 97 |
| SFC..... | 335 | periodic..... | 98 |
| REAL..... | 182 | TIME..... | 176 |
| Reference Data Type..... | 222 | timer | |
| reference declaration..... | 222 | event..... | 111 |
| REPEAT...UNTIL...END_REPEAT..... | 438 | TOD..... | 179 |
| RETURN | | U | |
| FBD..... | 266 | UDINT..... | 174 |
| IL..... | 392 | UINT..... | 174 |
| LD..... | 295 | W | |
| ST..... | 440 | warm start..... | 135 |
| S | | watchdogs | |
| S | | mono-task..... | 99 |
| IL..... | 384 | multi-task..... | 105 |
| LD..... | 282 | WHILE...DO...END_WHILE | |

| | |
|------------|-----|
| ST | 438 |
| WORD | 190 |

X

| | |
|----------|-----|
| XOR | |
| IL | 386 |
| ST | 429 |

Schneider Electric
35 rue Joseph Monier
92500 Rueil Malmaison
France

+ 33 (0) 1 41 29 70 00

www.se.com

As standards, specifications, and design change from time to time, please ask for confirmation of the information given in this publication.

© 2021 Schneider Electric. All rights reserved.

35006144.25