EcoStruxure Machine Expert

Programming Guide

EIO0000002854.10
10/2026

se.com

Legal Information

The information provided in this document contains general descriptions, technical characteristics and/or recommendations related to products/solutions.

This document is not intended as a substitute for a detailed study or operational and site-specific development or schematic plan. It is not to be used for determining suitability or reliability of the products/solutions for specific user applications. It is the duty of any such user to perform or have any professional expert of its choice (integrator, specifier or the like) perform the appropriate and comprehensive risk analysis, evaluation and testing of the products/solutions with respect to the relevant specific application or use thereof.

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

This document and its content are protected under applicable copyright laws and provided for informative use only. No part of this document 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 document or its content, except for a non-exclusive and personal license to consult it on an "as is" basis.

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.

Schneider Electric reserves the right to make changes or updates with respect to or in the content of this document or the format thereof, at any time 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 document, as well as any non-intended use or misuse of the content thereof.

Table of Contents

Table of Contents 3
Safety Information 17
About the Document 18
Operational Limitations of Agentic AI Engineering Systems 23
Introduction 27
General Introduction to the Logic Builder 28
What is the Logic Builder? 28
Tasks Performed by the Logic Builder 28
User Interface 30
Elements of the Logic Builder Screen 30
Multi-Tabbed **Navigators** 35
Functional tree 42
Multi-Tabbed **Catalog** View 45
Status & Staging View (exclusive to *.fbs projects) 46
Customizing the User Interface 46
User Interface in Online Mode 50
Menus and Commands 52
Basic Concepts 53
Introduction and Basic Concepts 53
Configuration 55
Installing Devices 56
Integration of Sercos Devices from Third-Party Vendors 56
Managing Devices 58
Adding Devices through the **Hardware Catalog** 58
Adding Devices by Drag and Drop 58
Adding Devices by Contextual Menu or Plus Button 60
Adding a Controller 60
Adding Expansion Devices 61
Adding Communication Managers 62
Adding Devices to a Communication Manager 63
Adding Devices from Template 66
Updating Devices 66
Updating Devices 66
Converting Devices 67
Converting Devices 67
Converting Projects 69
Converting SoMachine Basic and Twido Projects 69
Common Device Editor Dialogs 85
Device Configuration 85
General Information About Device Editors 85
Communication Settings in Controller Selection Mode 87
Communication Settings in Simple Mode 111
Communication Settings in Classic Mode 117
Configuration 121
Parameters 122
Applications 122
Synchronized Files 123
Files 124

Log 125
PLC Settings 128
IEC Objects 130
Users and Groups 132
Access Rights 137
Symbol Rights 142
Task Deployment 143
Status 144
Information 144
Emulation Settings 144
I/O Mapping 146
I/O Mapping 146
Working with the I/O Mapping Dialog Box 148
Mapping a Device and a Function Block Instance 153
I/O Mapping in Online Mode 154
Implicit Variables for Forcing I/Os 154
Select Function Block Dialog Box 155
Cybersecurity 157
General Information 158
Access Protection with Device User Management 158
Firewall Settings 158
Using Certificates 158
Digital Signature Verification 159
Digital Signature Verification 159
Connecting to the Controller 162
First Login to the Controller with User Rights Management Activated 162
Logoff Current Device User 163
Logoff / Change Device User 163
Security for the Project and the Application 164
Protecting and Saving a Project (exclusive to *.project and *.library files) 165
Protecting and Saving Applications 165
Security for the Runtime System / Controller 167
Encrypting Communication 167
Handling of Device User Management 167
Visualization 170
Visualization 170
FAQ 171
Frequently Asked Questions (FAQ) 171
Program 172
Program Components 173
Program Organization Unit (POU) 173
POU 173
Adding and Calling POU Objects 174
Program 177
Function 179
Method 181
Property 184
Interface 186
Action 188

Transition 190
POUs for Implicit Checks 191
Function Block 192
General Information 192
Function Block Instance 193
Calling a Function Block 194
Configuring a Memory Reserve for the Online Change of Function Blocks 195
Extension of a Function Block 197
Implementing Interfaces 198
Method Invocation 200
SUPER Pointer 201
THIS Pointer 203
Application Objects 204
Data Type Unit (DUT) 204
Global Variable List - GVL 205
Global Variable List - GVL (tasklocal) 207
Network Variable List (Receiver) 212
Persistent Variables 218
External File 220
Text List 222
Image Pool 228
Application 232
Application 232
Task Configuration 233
Task Configuration 233
Adding Tasks 233
Managing Applications 234
General Information 234
Introduction 234
Building and Downloading Applications 235
Building Applications 235
Login 237
Build Process at Changed Applications 238
Downloading an Application 239
Managing Expired Certificates 249
Running Applications 250
Running Applications 250
Maintaining Applications 251
Monitoring 251
Debugging 251
Core Dump 256
Programming Support 258
Refactoring 258
Static Analysis Light 262
Creating an Archive in Your Controller on Download 264
Logic Editors 265
Common Features of Graphic Editors 266
Common Features of Graphic Editors 266
FBD/LD/IL Editor 268
Information on the FBD/LD/IL Editor 268

FBD/LD/IL Editor ... 268
Function Block Diagram (FBD) Language ... 268
Ladder Diagram (LD) Language ... 269
Instruction List (IL) Language ... 269
Modifiers and Operators in IL ... 270
Working in the FBD and LD Editor ... 274
Working in the IL Editor ... 278
Cursor Positions in FBD, LD, and IL ... 283
FBD/LD/IL Menu ... 286
FBD/LD/IL Editor in Online Mode ... 287
FBD/LD/IL Elements ... 291
FBD/LD/IL Toolbox ... 291
Network in FBD/LD/IL ... 292
Assignment in FBD/LD/IL ... 293
Jump in FBD/LD/IL ... 294
Label in FBD/LD/IL ... 294
Boxes in FBD/LD/IL ... 294
RETURN Instruction in FBD/LD/IL ... 295
Branch / Hanging Coil in FBD/LD/IL ... 295
Parallel Branch ... 297
Branch Start / End ... 299
Set/Reset in FBD/LD/IL ... 299
Set/Reset Coil ... 300
Execute ... 300
LD Elements ... 301
Contact ... 301
Coil ... 302
Ladder Editor ... 303
General Information on the Ladder Editor ... 303
Programming in the Ladder Editor ... 304
Monitoring and Detecting Errors ... 307
Navigating in the Ladder Editor ... 309
Continuous Function Chart (CFC) Editor ... 310
Continuous Function Chart (CFC) Language ... 310
CFC Editor ... 310
Cursor Positions in CFC ... 312
CFC Elements / ToolBox ... 314
Working in the CFC Editor ... 319
Execution Order in CFC ... 322
CFC Editor in Online Mode ... 325
CFC Editor Page-Oriented ... 328
Sequential Function Chart (SFC) Editor ... 330
SFC Editor ... 330
SFC - Sequential Function Chart Language ... 331
Cursor Positions in SFC ... 332
Working in the SFC Editor ... 333
SFC Element Properties ... 335
SFC Elements / ToolBox ... 336
Qualifier for Actions in SFC ... 346
Implicit Variables - SFC Flags ... 347
Sequence of Processing in SFC ... 353

SFC Editor in Online Mode 354
Structured Text (ST) Editor 356
Information on the ST Editor 356
ST Editor 356
ST Editor in Online Mode 357
Structured Text ST 360
Structured Text ST 360
Expressions 360
Instructions 362
Object Editors 372
Declaration Editors 373
Textual Declaration Editor 373
Tabular Declaration Editor 374
Declaration Editor in Online Mode 378
Device Type Manager (DTM) Editor 380
DTM Editor 380
Data Unit Type (DUT) Editor 381
Data Unit Type Editor 381
Global Variables List (GVL) Editor 382
GVL Editor 382
Motion Design Object Editor 383
Motion Design Object Editor 383
Network Variables List (NVL) Editor 386
Information on the NVL Editor 386
Network Variables List Editor 386
General Information on Network Variables 387
Introduction to Network Variables List (NVL) 387
Configuring the Network Variables Exchange 390
Network Variables List (NVL) Rules 394
Operating State of the Sender and the Receiver 396
Example 397
Compatibility 400
Task Editor 404
Information on the Task Configuration 404
Task Groups Tab 405
Properties Tab 406
System Events Tab 407
Monitor Tab 409
Variable Usage Tab 410
Stack Usage Tab 411
Configuration of a Specific Task 413
Task Processing in Online Mode 416
Multicore 417
Multicore - Tools 420
CPU Load Tab 422
Watch List Editor 423
Watch View / Watch List Editor 423
Creating a Watch List 423
Watch List in Online Mode 424
Tools Within Logic Editors 426
Function and Function Block Finder 426

Input Assistant ... 429
Tools ... 431
Data Logging ... 432
Introduction to Data Logging ... 432
Recipe Manager ... 434
Recipe Manager ... 435
Recipe Definition ... 439
RecipeManCommands ... 444
Loading Recipe Values from the Controller ... 452
Memory Usage by Recipes ... 453
Trace Editor ... 454
Trace Object ... 454
Trace Basics ... 454
Creating a Trace or a Device Trace Object ... 456
Trace Configuration ... 460
Trace Configuration - Tree Views ... 460
Variable Settings ... 462
Record Settings ... 464
Display Mode ... 466
Advanced Trace Settings ... 468
Trace Editor in Online Mode ... 469
Trace Editor in Online Mode ... 469
Keyboard Operations for Trace Diagrams ... 471
Keyboard Shortcuts ... 471
Trend Recording ... 472
Trend Recording Objects ... 472
Trend Recording Overview ... 472
Trend Recording Objects ... 473
Trend Recording Configuration ... 474
Trend Recording Editor ... 474
Record Settings ... 474
Variable Settings ... 476
Steps for Trend Recording Configuration ... 478
Recording Trend Data ... 480
Start the Recording Process ... 480
Alarm Configuration ... 481
General Information ... 481
Alarm Management ... 481
Defining Alarms ... 482
Configuration of the Alarm Management ... 483
Creating an Alarm Configuration ... 483
Configuring an Alarm Class ... 483
Configuring an Alarm Storage ... 484
Configuring an Alarm Group ... 485
Calling Event Alarms in the Program ... 486
Defining an **Event** Alarm in the Alarm Group ... 486
Program Call for Triggering the **Event** Alarm ... 486
Alarm Management Objects ... 487
Alarm Configuration ... 487
Alarm Group Template ... 487
Alarm Class ... 490

Alarm Group ... 495
Alarm Storage ... 498
Unit Conversion ... 501
Unit Conversion Configuration ... 501
Usage in IEC Editors ... 503
Symbol Configuration Editor ... 505
Symbol Configuration Editor ... 505
Symbol Configuration ... 511
Adding a Symbol Configuration ... 512
Data Exchange: Controller - HMIs, OPC DA and UA Server ... 517
Configuring Data Exchange of Variables ... 517
Publishing Variables in the Controller ... 521
Selecting Variables in the HMI ... 522
Publishing Variables in the HMI of Vijeo-Designer ... 523
Parameterization of the Physical Media ... 524
Data Exchange Performance Between Controller and HMI Configured with Vijeo-Designer ... 525
Indications for HMI Connections with Vijeo-Designer ... 527
Cam Motion Editor ... 529
Cam Motion Editor - General Information ... 529
General Information ... 529
Adding a Cam Object ... 529
Opening the **Motion Editor** of a Cam Object ... 530
Generating IEC Program Code from Cam Data ... 530
Using Cam Data for Function Blocks ... 530
Copying the Source Code of the Cam Diagram ... 532
Online View and Functions of the Cam Motion Editor ... 532
Online View and Functions of the Cam Motion Editor ... 532
Discontinuous Courses of the Positions ... 533
Discontinuous Courses of the Positions ... 533
Dialog Boxes ... 534
Motion Editor ... 534
Generating IEC Source Code ... 544
Configuration ... 545
Script Hook Manager ... 546
Script Hook Manager ... 546
Programming Reference ... 549
Variables Declaration ... 550
Declaration ... 550
General Information ... 550
Best Practices for the Naming of Identifiers ... 553
Shadowing Rules ... 557
Variables Initialization ... 560
Declaration ... 561
Shortcut Mode ... 561
`AT` Declaration ... 562
Keywords ... 563
Variable Types ... 566
Variable Types ... 566
Attribute Keywords for Variable Types ... 570
`VAR_CONFIG` - Configuration Variables ... 578

Method Types 579
FB_Init, *FB_Reinit*, and *FB_Exit* Methods 579
Pragma Instructions 583
Pragma Instructions 583
Message Pragmas 585
Conditional Pragmas 587
Region Pragmas 597
Attribute Pragmas 597
Attribute Pragmas 597
User-Defined Attributes 597
`Attribute call_after_global_init_slot` 598
`Attribute call_after_init` 599
`Attribute call_after_online_change_slot` 599
`Attribute call_before_global_exit_slot` 600
Attribute call_on_type_change 600
`Attribute conditionalshow` 601
`Attribute conditionalshow_all_locals` 603
`Attribute const_replaced`, `Attribute const_non_replaced` 603
`Attribute 'dataflow'` 604
`Attribute displaymode` 605
`Attribute enable_dynamic_creation` 605
`Attribute estimated-stack-usage` 606
`Attribute ExpandFully` 606
`Attribute global_init_slot` 607
`Attribute hide` 609
`Attribute hide_all_locals` 610
`Attribute initialize_on_call` 611
`Attribute init_namespace` 611
`Attribute init_On_Onlchange` 612
`Attribute instance-path` 612
`Attributes io_function_block`, `io_function_block_mapping` 613
`Attribute is_connected` 614
`Attribute linkalways` 614
`Attribute monitoring` 615
`Attribute monitoring_display` 618
`Attribute monitoring_encoding` 618
`Attribute namespace` 619
`Attributes no_assign`, `no_assign_warning` 620
`Attribute no_check` 621
`Attribute no_copy` 622
`Attribute no-exit` 622
`Attribute no_init` 623
`Attribute no_instance_in_retain` 623
`Attribute no_virtual_actions` 623
`Attribute pingroup` 625
`Attribute pin_presentation_order_inputs/outputs` 626
`Attribute obsolete` 627
`Attribute pack_mode` 628

Attribute ProcessValue 629
Attribute qualified_only 629
Attribute reflection 630
Attribute subsequent 630
Attribute suppress_warning 631
Attribute symbol 631
Attribute to_string 632
Attribute warning disable, warning restore 633
The Smart Coding Functionality 634
Smart Coding 634
Data Types 635
General Information 635
Data Types 635
Standard Data Types 635
Standard Data Types 635
UTF-8 Encoding 642
Extensions to IEC Standard 644
UNION 644
BIT 645
References 646
Pointers 647
User-Defined Data Types 650
Defined Data Types 650
Arrays 651
Vector 654
Structures 661
Enumerations 663
Implicit Enumerations 665
Subrange Types 666
Alias 668
VERSION 670
Programming Guidelines 671
Naming Conventions 671
General Information 671
Prefixes 672
Prefix Parts 672
Order of Prefixes 672
Scope Prefix 673
Data Type Prefix 674
Property Prefix 675
POU Prefix 676
Namespace Prefix 676
SmartTag Functions 677
SmartTag Function 677
Operators 678
Arithmetic Operators 678
ADD 679
MUL 681
SUB 682
DIV 683
MOD 685

MOVE ... 686
SIZEOF ... 687
XSIZEOF ... 688
Bitstring Operators ... 688
AND ... 689
OR ... 689
XOR ... 690
NOT ... 691
Bit-Shift Operators ... 691
SHL ... 691
SHR ... 692
ROL ... 693
ROR ... 694
Selection Operators ... 695
SEL ... 695
MAX ... 697
MIN ... 697
LIMIT ... 698
MUX ... 698
Comparison Operators ... 699
GT ... 699
LT ... 700
LE ... 700
GE ... 701
EQ ... 702
NE ... 702
Address Operators ... 703
ADR ... 703
Content Operator ... 704
BITADR ... 704
Calling Operator ... 705
CAL ... 705
Type Conversion Operators ... 706
Type Conversion Functions ... 706
Overloaded Conversions ... 707
BOOL_TO Conversions ... 708
TO_BOOL Conversions ... 709
Conversion Between Integral Number Types ... 711
REAL_TO / LREAL_TO Conversions ... 711
TIME_TO/TIME_OF_DAY Conversions ... 712
DATE_TO/DT_TO Conversions ... 714
STRING_TO Conversions ... 715
TRUNC ... 716
TRUNC_INT ... 717
ANY_..._TO Conversions ... 718
TO_<xxx> Conversions ... 718
Numeric Functions ... 719
ABS ... 719
SQRT ... 719
LN ... 720
LOG ... 720

EXP ... 721
SIN ... 721
COS ... 722
TAN ... 723
ASIN ... 723
ACOS ... 724
ATAN ... 724
EXPT ... 725
IEC Extending Operators ... 725
IEC Extending Operators ... 725
__DELETE ... 726
__ISVALIDREF ... 727
__NEW ... 728
__QUERYINTERFACE ... 730
__QUERYPOINTER ... 731
AND_THEN ... 732
OR_ELSE ... 732
__TRY, __CATCH, __FINALLY, __ENDTRY ... 733
__*VARINFO* ... 736
'__POSITION' ... 737
'__POUNAME' ... 738
__POOL ... 738
Scope Operators ... 739
Initialization Operator ... 741
INI Operator ... 741
Multicore Operators ... 742
__CURRENTTASK ... 742
__COMPARE_AND_SWAP ... 743
__XADD ... 744
TEST_AND_SET ... 745
Operands ... 746
Constants ... 746
BOOL Constants ... 746
TIME/LTIME Constants ... 746
DATE Constants ... 747
LDATE Constants ... 747
DATE_AND_TIME Constants ... 748
LDATE_AND_TIME Constants ... 748
TIME_OF_DAY Constants ... 749
LTIME_OF_DAY Constants ... 750
Number Constants ... 750
REAL/LREAL Constants ... 751
String Constants ... 751
Character Constant UCHAR ... 753
Typed Constants / Typed Literals ... 753
Variables ... 754
Variables ... 754
Addressing Bits in Variables ... 755
Partial Variable Access ... 757
Addresses ... 760
Direct Addresses ... 760

Functions 761
Functions 761
Templates 762
General Information about Templates 763
Templates 763
General Information About Templates 763
Administration of Templates 764
Managing Device Templates 770
Managing Device Templates 770
Facts of Device Templates 770
Adding Devices from Template 770
Creating a Device Template on the Basis of Field Devices or I/O Modules 773
Visualizations Suitable for Creating Device Templates 773
Further Information on Integrating Control Logic into Device Templates 774
Steps to Create a Device Template 776
Managing Function Templates 779
Managing Function Templates 779
Facts of Function Templates 779
Adding Functions from Template 780
Application Functions as Basis for Function Templates 784
Steps to Create a Function Template 785
Troubleshooting and FAQ 790
Generic - Troubleshooting and FAQ 791
Troubleshooting 791
Log File 791
Frequently Asked Questions 791
How Can I Enable and Configure Analog Inputs on CANopen? 791
Why is the Startup Performance Sometimes Slower? 792
How Can I Manage Shortcuts and Menus? 793
How Can I Increase the Build-Time Performance? 794
What Can I Do in Case of Issues with Modbus IOScanner on Serial Line? 794
What Can I Do If My Network Variables List (NVL) Communication Has Been Suspended? 795
Accessing Controllers - Troubleshooting and FAQ 796
Troubleshooting: Accessing New Controllers 796
Accessing New Controllers 796
Connecting via IP Address and Address Information 798
FAQ - What Can I Do in Case of Connection Problems With the Controller? 799
FAQ - Why is a Connection to the Controller not Possible? 799
FAQ - Why has the Communication Between PC and Controller been Interrupted? 802
Appendices 803
Network Communication 804
Network Topology 804
Addressing and Routing 804

Structure of Addresses ... 806
Python Script Language ... 809
General Information ... 809
Introduction ... 809
Accessing the Python Interpreter ... 811
Using the Logic Builder Shell ... 813
Using the Logic Builder **Scripting Immediate** View ... 819
Keyboard Commands in the Logic Builder Shell and the **Scripting Immediate** View ... 821
Explore Python API (with `dir()` and `inspectapi`) ... 822
Executing Scripts ... 826
Best Practices ... 827
EcoStruxure Machine Expert Python API ... 829
EcoStruxure Machine Expert Scripting - Python API ... 830
Calling Scripts Via Toolbar Icons ... 834
Schneider Electric Script Engine Examples ... 836
New Projects ... 836
Device Parameters ... 837
Compiler Version ... 838
Update Project ... 839
Update Libraries ... 839
Clean and Build Application ... 840
Communication Settings ... 841
Start ETEST ... 841
Reset Diagnostic Messages ... 843
Reboot the Controller ... 843
Convert Device ... 844
Comparing Projects ... 846
Advanced Library Management Functions ... 847
Accessing POUs ... 847
CODESYS Script Engine Examples ... 848
Project ... 849
Online Application ... 855
Objects ... 857
Devices ... 858
System / User Interface (UI) ... 860
Reading Values ... 863
Reading Values From Recipe and Send an Email ... 864
Determine Device Tree of the Open Project ... 865
Script Example 4: Import a Device in PLCOpenXML From Subversion ... 866
Script Example 5: Creating and Editing POUs ... 867
Script Example 6: User Interface / Interaction with the User ... 868
Script Example 7: Manipulation of the **Project Information** Object ... 869
Script Example 8: Trace Scripting ... 870
Advanced Example: Checkout a Library from SVN and Installation ... 874
Managing Device User Rights Using the Scripting API ... 874
Managing Device User Rights Using the Scripting API ... 874
Controller Feature Sets for Migration ... 876

Controller Feature Sets for Migration 876
Glossary 879
Index 883

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.

About the Document

Document Scope

This document describes the graphical user interface of the EcoStruxure Machine Expert software and the functions they provide. For further information, refer to the separate documents provided in the respective online help.

Validity Note

This document has been updated for the release of EcoStruxure™ Machine Expert V2.6.1.

Product Related Information

⚠WARNING

LOSS OF CONTROL

- Perform a Failure Mode and Effects Analysis (FMEA), or equivalent risk analysis, of your application, and apply preventive and detective controls before implementation.
- Provide a fallback state for undesired control events or sequences.
- Provide separate or redundant control paths wherever required.
- Supply appropriate parameters, particularly for limits.
- Review the implications of transmission delays and take actions to mitigate them.
- Review the implications of communication link interruptions and take actions to mitigate them.
- Provide independent paths for control functions (for example, emergency stop, over-limit conditions, and error conditions) according to your risk assessment, and applicable codes and regulations.
- Apply local accident prevention and safety regulations and guidelines.[1]
- Test each implementation of a system for proper operation before placing it into service.

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

[1] For additional information, refer to NEMA ICS 1.1 (latest edition), *Safety Guidelines for the Application, Installation, and Maintenance of Solid State Control* and to NEMA ICS 7.1 (latest edition), *Safety Standards for Construction and Guide for Selection, Installation and Operation of Adjustable-Speed Drive Systems* or their equivalent governing your particular location.

⚠WARNING

UNINTENDED EQUIPMENT OPERATION

- Only use software approved by Schneider Electric for use with this equipment.
- Update your application program every time you change the physical hardware configuration.

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

For reasons of Internet security, for those devices that have a native Ethernet connection, TCP/IP forwarding is disabled by default. Therefore, you must manually enable TCP/IP forwarding. However, doing so may expose your network to possible cyberattacks if you do not take additional measures to protect your enterprise. In addition, you may be subject to laws and regulations concerning cybersecurity.

⚠WARNING

UNAUTHENTICATED ACCESS AND SUBSEQUENT NETWORK INTRUSION

- Observe and respect any and all pertinent national, regional and local cybersecurity and/or personal data laws and regulations when enabling TCP/IP forwarding on an industrial network.
- Isolate your industrial network from other networks inside your company.
- Protect any network against unintended access by using firewalls, VPN, or other, proven security measures.

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

Consult the Schneider Electric Cybersecurity Best Practices for additional information.

General Cybersecurity Information

In recent years, the growing number of networked machines and production plants has seen a corresponding increase in the potential for cyber threats, such as unauthorized access, data breaches, and operational disruptions. You must, therefore, consider all possible cybersecurity measures to help protect assets and systems against such threats.

To help keep your Schneider Electric products secure and protected, it is in your best interest to implement the cybersecurity best practices as described in the Cybersecurity Best Practices document.

Schneider Electric provides additional information and assistance:

- Subscribe to the Schneider Electric security newsletter.
- Visit the Cybersecurity Support Portal web page to:
  - Find Security Notifications.
  - Report vulnerabilities and incidents.
- Visit the Schneider Electric Cybersecurity and Data Protection Posture web page to:
  - Access the cybersecurity posture.
  - Learn more about cybersecurity in the cybersecurity academy.
  - Explore the cybersecurity services from Schneider Electric.

Product Related Cybersecurity Information

Refer to the EcoStruxure Machine Expert Cybersecurity Guide.

NOTE: Schneider Electric adheres to industry best practices in the development and implementation of control systems. This includes a "Defense-in-Depth" approach to secure an Industrial Control System. This approach places the controllers behind one or more firewalls to restrict access to authorized personnel and protocols only.

⚠WARNING

UNAUTHENTICATED ACCESS AND SUBSEQUENT UNAUTHORIZED MACHINE OPERATION

- Evaluate whether your application environments are connected to your critical infrastructure and, if so, take appropriate steps in terms of prevention, based on Defense-in-Depth, before connecting the automation system to any network.
- Limit the number of devices connected to a network to the minimum necessary.
- Isolate your industrial network from other networks inside your company.
- Protect any network against unintended access by using firewalls, VPN, or other, proven security measures such as an Intrusion Prevention System or Intrusion Detection System.
- Monitor activities within your systems.
- Prevent subject devices from direct access or direct link by unauthorized parties or unauthenticated actions.
- Install certificates that are issued by publicly known Trusted Certificate Authorities.
- Keep your systems up to date and rely only on legitimate sources.
- Prepare a recovery plan including backup of your system and process information.

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

For more information on organizational measures and rules covering access to infrastructures, refer to ISO/IEC 27000 series, Common Criteria for Information Technology Security Evaluation, ISO/IEC 15408, IEC 62351, ISA/IEC 62443, NIST Cybersecurity Framework, Information Security Forum - Standard of Good Practice for Information Security and refer to EcoStruxure Machine Expert Cybersecurity Guide.

Refer to the specific Cybersecurity related information in the part Cybersecurity, page 157.

Related Documents

Document title	Reference
Cybersecurity Best Practices	CS-Best-Practices-2019-340
EcoStruxure Machine Expert Cybersecurity Guide	EIO0000004242 (ENG)
EcoStruxure Machine Expert Introduction	EIO0000002836 (ENG)
EcoStruxure Machine Expert Compatibility and Migration User Guide	EIO0000002842 (ENG)
EcoStruxure Machine Expert Functions and Libraries User Guide	EIO0000002829 (ENG)
EcoStruxure Machine Expert Device Type Manager (DTM) User Guide	EIO0000003047 (ENG) EIO0000003048 (FRE) EIO0000003049 (GER) EIO0000003050 (SPA) EIO0000003051 (ITA) EIO0000003052 (CHS)
TwidoEmulationSupport Library Guide	EIO0000001692 (ENG) EIO0000001693 (FRE) EIO0000001694 (GER) EIO0000001696 (SPA) EIO0000001695 (ITA) EIO0000001697 (CHS)
EcoStruxure Machine Expert Network Variable Configuration SE_NetVarUdp Library Guide	EIO0000002974 (ENG) EIO0000002975 (FRE) EIO0000002976 (GER) EIO0000002978 (SPA) EIO0000002977 (ITA) EIO0000002979 (CHS)

To find documents online, visit the Schneider Electric download center (www.se.com/ww/en/download/).

Information on Non-Inclusive or Insensitive Terminology

As a responsible, inclusive company, Schneider Electric is constantly updating its communications and products that contain non-inclusive or insensitive terminology. However, despite these efforts, our content may still contain terms that are deemed inappropriate by some customers.

Terminology Derived from Standards

The technical terms, terminology, symbols and the corresponding descriptions in the information contained herein, or that appear in or on the products themselves, are generally derived from the terms or definitions of international standards.

In the area of functional safety systems, drives and general automation, this may include, but is not limited to, terms such as *safety*, *safety function*, *safe state*, *fault*, *fault reset*, *malfunction*, *failure*, *error*, *error message*, *dangerous*, etc.

Among others, these standards include:

Standard	Description
IEC 61131-2:2007	Programmable controllers, part 2: Equipment requirements and tests.
ISO 13849-1:2023	Safety of machinery: Safety related parts of control systems. General principles for design.
EN 61496-1:2020	Safety of machinery: Electro-sensitive protective equipment. Part 1: General requirements and tests.
ISO 12100:2010	Safety of machinery - General principles for design - Risk assessment and risk reduction
EN 60204-1:2006	Safety of machinery - Electrical equipment of machines - Part 1: General requirements
ISO 14119:2013	Safety of machinery - Interlocking devices associated with guards - Principles for design and selection
ISO 13850:2015	Safety of machinery - Emergency stop - Principles for design
IEC 62061:2021	Safety of machinery - Functional safety of safety-related electrical, electronic, and electronic programmable control systems
IEC 61508-1:2010	Functional safety of electrical/electronic/programmable electronic safety-related systems: General requirements.
IEC 61508-2:2010	Functional safety of electrical/electronic/programmable electronic safety-related systems: Requirements for electrical/electronic/programmable electronic safety-related systems.
IEC 61508-3:2010	Functional safety of electrical/electronic/programmable electronic safety-related systems: Software requirements.
IEC 61784-3:2021	Industrial communication networks - Profiles - Part 3: Functional safety fieldbuses - General rules and profile definitions.
2006/42/EC	Machinery Directive
2014/30/EU	Electromagnetic Compatibility Directive
2014/35/EU	Low Voltage Directive

In addition, terms used in the present document may tangentially be used as they are derived from other standards such as:

Standard	Description
IEC 60034 series	Rotating electrical machines
IEC 61800 series	Adjustable speed electrical power drive systems
IEC 61158 series	Digital data communications for measurement and control – Fieldbus for use in industrial control systems
IEC 62443 series	Security for industrial automation and control systems

Finally, the term *zone of operation* may be used in conjunction with the description of specific hazards, and is defined as it is for a *hazard zone* or *danger zone* in the *Machinery Directive* (*2006/42/EC*) and *ISO 12100:2010*.

NOTE: The aforementioned standards may or may not apply to the specific products cited in the present documentation. For more information concerning the individual standards applicable to the products described herein, see the characteristics tables for those product references.

Operational Limitations of Agentic AI Engineering Systems

Agentic Artificial Intelligence engineering systems, including generative and retrieval-based models, are increasingly used to support engineering workflows. However, these systems have fundamental limitations that can lead to errors, logical flaws, or unsafe outputs — even when relevant data exists in the training data or knowledge base, or verified by the Artificial Intelligence (AI) using ETEST or the compiler.

This Agentic AI is intended solely as a productivity tool to support experienced automation engineers. It does not replace professional engineering judgment, formal design reviews, or mandatory verification and validation processes required for industrial control systems.

AI-generated code may contain subtle logical errors, race conditions, incorrect timer/counter behavior, unsafe state transitions, or improper handling of error conditions that are not obvious on casual inspection.

All outputs generated by the Agentic AI, particularly those involving automation or control code, must be reviewed and validated through engineering quality procedures before use in production environments.

⚠ WARNING

UNINTENDED EQUIPMENT OPERATION

- Do not include executable code delivered by Agentic AI engineering systems directly in an operational machine or process.
- All Agentic AI-generated Program Organization Units (POUs), function blocks, variables, or control logic must undergo thorough review, simulation, testing, and validation according to your quality procedures and applicable standards.
- Always perform a safety-related analysis (formal risk assessment and SIL/PL determination) for the complete application and all connected devices.
- Provide independent safety functions (e.g., hardwired emergency stops, safety relays, or certified safety controller logic) that do not rely on Agentic AI-generated code.
- Verify full compatibility with the exact controller reference, firmware version, hardware configuration, I/O mapping, and libraries used in your application/system.

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

The accuracy, contextual understanding, and overall reliability of the Agentic AI is continuously improved over time. However, iterative improvements do not eliminate the fundamental limitations described below, nor do they remove the requirement for full engineering validation of every output.

1. **Variability in Responses (Non-Deterministic Behavior)**
   - **Description**: Identical prompts may produce different outputs across sessions, including variations in generated control logic.
   - **Cause**: Agentic AI relies on probabilistic generation, which can yield multiple, seemingly valid but behaviorally different interpretations.
   - **Controller-specific risk:** Even small differences in code structure, condition ordering, or error handling can lead to inconsistent machine behavior or unintended safety gaps.
   - **Best Practice**: For repeatable or safety-critical tasks, use highly structured and constrained prompts. Generate and compare multiple versions, then document and version-control only the validated output.
2. **Factual Inaccuracies and Logical Errors in Code**
   - **Description**: The Agentic AI may generate inaccurate incomplete, or logically inconsistent information and code, even when correct data exists in the knowledge base or training material.
   - **Cause**: Generative models predict plausible responses rather than performing true engineering reasoning or formal verification. They can misinterpret controller-specific instructions, timing behavior, or safety requirements.
   - **Best Practice**: Always validate outputs against the latest official Schneider Electric documentation and the established standards and coding guidelines of your organization. Use precise, context-rich prompts to help improve reliability. Never rely on the Agentic AI for safety-related logic.
3. **Incomplete, Incompatible, or Non-Optimal Code**
   - **Description**: Code generated or retrieved may be incomplete or incompatible with your specific Schneider Electric system configuration, or transgress hardware constraints (memory limits, scan cycle time, I/O addressing, etc.).
   - **Cause**: Limited context or partial matches in the retrieval process.
   - **Best Practice**: Always include system-specific information in your prompts (controller reference, firmware version, relevant hardware configuration, etc.). Manually review, complete, adapt, and test all generated code in the Schneider Electric development environment (e.g., EcoStruxure Machine Expert). Perform compilation and resource usage verifications before any deployment.
4. **Outdated or Irrelevant Retrievals**
   - **Description**: The Agentic AI may retrieve or generate content based on legacy documents, deprecated instructions, or practices that no longer comply with current standards.
   - **Cause**: The knowledge base includes historical data that may not reflect present-day standards.
   - **Best Practice**: Always cross-check against the most recent official Schneider Electric documentation and applicable standards. Report any outdated content via feedback for review and updating.
5. **Conflicting Information Across Sources**
   - **Description**: Responses may include contradictory statements or code examples.
   - **Cause**: The Agentic AI integrates content from multiple Schneider Electric sources, which may reflect differing practices or legacy standards.
   - **Best Practice**: Use the most recent Schneider Electric documentation. Consult your engineering guidelines when discrepancies arise.
6. **Hallucinated or False Safety-Critical Behavior**

- **Description:** The Agentic AI may generate code that appears to implement safety functions (e.g., emergency stop handling, guard monitoring, safe torque off) but contains hidden logical flaws.
- **Cause:** The model lacks the ability to perform formal functional safety analysis, reference verification, or exhaustive error scenario evaluation.
- **Controller-specific risk:** A false sense of security; the code may pass basic tests but fail under real error conditions or specific timing scenarios.
- **Best Practice:** Never use Agentic AI-generated code for any safety-related functions. All safety logic must be developed, verified, and validated exclusively by qualified safety engineers using certified tools and methodologies.

7. **Lack of Formal Verification and Certification**
   - **Description:** Agentic AI systems cannot provide mathematically proven correctness, exhaustive state-space analysis, or compliance with functional safety certification requirements.
   - **Best Practice:** For applications requiring Safety Integrity Level (SIL) 2 or higher, or Performance Level (PL) d/e, rely solely on traditional engineering processes, certified safety controllers, and validated function blocks. Agentic AI and its outputs are not certified for use in safety-related systems.

Conclusion and User Responsibility

The Agentic AI is designed as a supportive productivity tool to assist experienced automation engineers in generating code, ETEST, devices as described in Commands of the **Agentic AI** Menu in the Menu Commands online help. Despite continuous improvements, it remains subject to the inherent limitations outlined above. Users bear sole responsibility for the review, adaptation, verification, validation, and safe deployment of any AI-generated content.

All outputs — especially control logic, Program Organization Units (POUs), and safety-critical code — must be rigorously tested and validated using established engineering processes, the latest official Schneider Electric documentation, and applicable safety standards before use in any machine or process.

Legal Disclaimer

This feature is powered by Artificial Intelligence and generates automated responses that may not be always accurate, complete or up to date. Do not rely solely on this information for decision making, including for real-time control, safety critical decision-making, or emergency response in industrial environments and read carefully the safety instructions as indicated in the documentation.

By using the Agentic AI, you acknowledge and accept these operational limitations and agree that Schneider Electric shall not be liable for any damages or losses arising from the use or misuse of Agentic AI-generated outputs.

Introduction

What’s in This Part

General Introduction to the Logic Builder.....28
User Interface.....30
Basic Concepts.....53

General Introduction to the Logic Builder

What's in This Chapter

What is the Logic Builder? 28
Tasks Performed by the Logic Builder 28

What is the Logic Builder?

General Description

The Logic Builder provides the configuration and programming environment for projects.

It displays the different elements of your project in separate views that you can arrange on the user interface and on your desktop according to your individual requirements. This view structure allows you to add hardware and software elements to your project by drag and drop. The main configuration dialog boxes that allow you to create content for the project are provided in the center of the Logic Builder screen.

In addition to easy configuration and programming, the Logic Builder also provides powerful diagnostic and maintenance features.

Tasks Performed by the Logic Builder

Configuring and Programming Projects

The Logic Builder allows you to program logic and add devices to projects.

To assist you in performing this task, it provides the following functions:

- Separate hardware catalog views for **Controller, HMI & iPC, Devices & Modules, Diverse** allow you to add hardware devices to your project by drag and drop. It also allows you to use device templates and function templates.
- A separate software catalog view for **ToolBox** allows you to add software elements by drag and drop.

To display only the relevant views for the task that is being performed, individual perspectives, page 49 are provided for hardware configuration, software configuration, and online mode. You are allowed to adapt these default perspectives to your individual requirements, and to create your own perspectives with the views you use most frequently.

Building Projects

The Logic Builder provides different ways (such as **Generate Code** or **Generate Code for all Applications**) to build your project.

Communication with Controller

The Logic Builder provides scan functions to detect available controllers in the Ethernet network. It supports different protocols for communication with the controller.

After communication has been established, applications can be downloaded to or uploaded from the controller. Applications can be started and stopped on the controller.

Online Features

The Logic Builder online features allow you to perform the following tasks:

- Online monitoring of values in program code and in **Watch** views
- Performing online changes
- Online configuration of traces
- Watching traces online
- Interacting with your machine by using built-in visualizations in online mode for diagnostic and test purposes
- Reading the status of controllers and devices
- Detecting potential programming logic errors by using the debugging function

User Interface

What's in This Chapter

Elements of the Logic Builder Screen 30
Multi-Tabbed **Navigators** 35
Functional tree 42
Multi-Tabbed **Catalog** View 45
Status & Staging View (exclusive to *.fbs projects) 46
Customizing the User Interface 46
User Interface in Online Mode 50
Menus and Commands 52

Elements of the Logic Builder Screen

Overview

Logic Builder consists of the following elements:

- Menus and toolbars
- **Navigator** views
- **Catalog** views
- Main editor pane

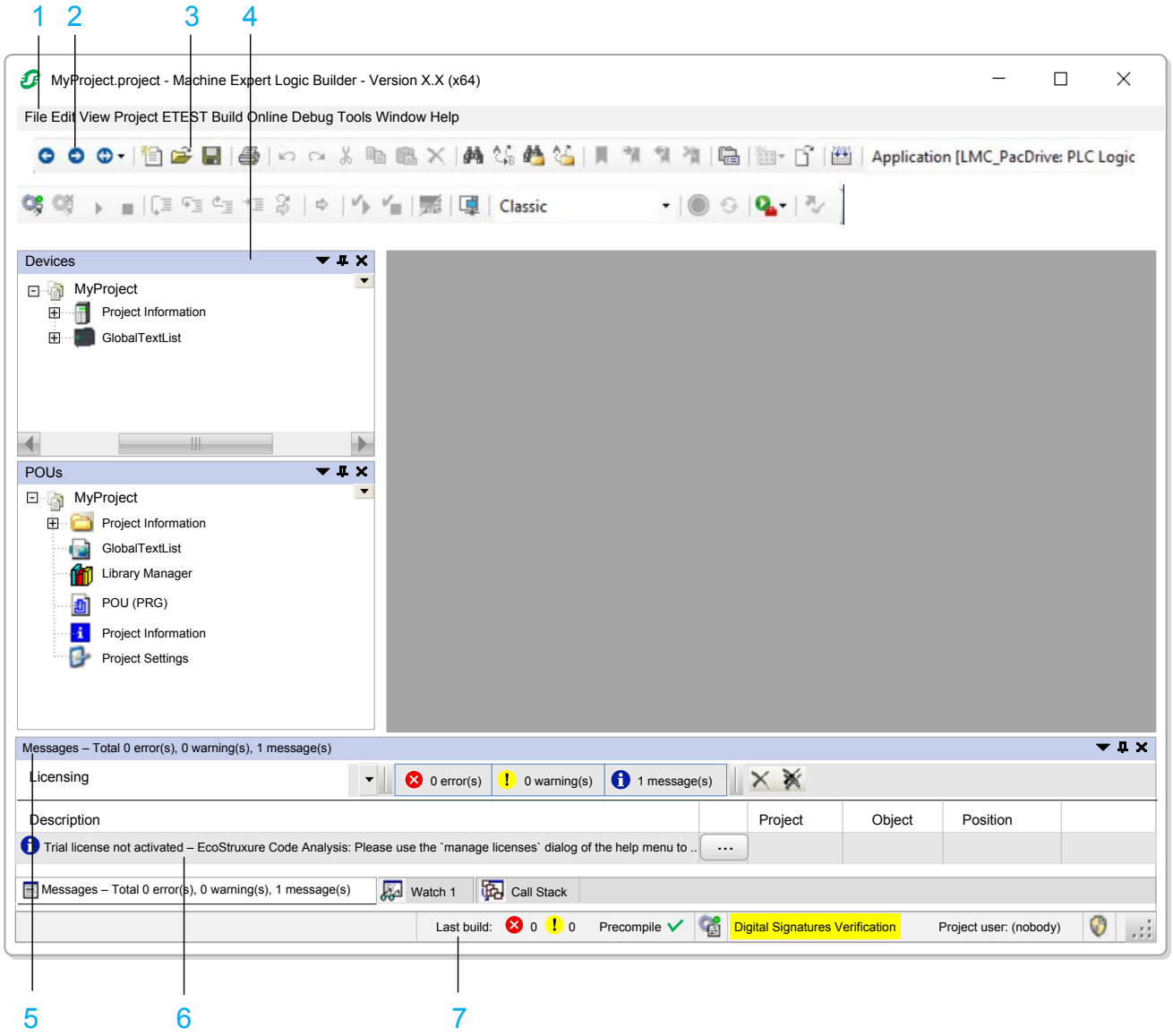
When you open the Logic Builder, it provides a default screen layout. This document describes the default positions.

You can adapt the elements according to your individual requirements as described in the *Customizing the User Interface* chapter, page 46. You can see and modify the present settings in the **Customize** dialog box. It is by default available in the **Tools** menu.

You can also arrange the views and windows by shifting, docking/undocking views, resizing or closing windows. The positions are saved with the project. When you reopen a project, the elements are placed at the positions where they were when the project was saved. The positions of views are saved separately in perspectives, page 49.

Default Logic Builder Screen

Default positions of menus, bars, and views on the Logic Builder screen



1 Menu bar

2 Buttons for navigation within editors

3 Toolbar

4 Multi-tabbed **Navigators**: **Devices tree, Tools tree, Applications tree, Functional tree**

5 Messages view

6 Information bar

7 Status bar

Default Components

The Logic Builder screen contains the following components that are visible by default:

Component	Description
Menu bar	Provides menus which contain the available commands as defined in the **Tools > Customize** dialog box.
Buttons for navigation within editors	Three buttons for navigating within editor views: • **Navigate backward**: Allows you to jump to the last cursor position within an editor view. As a history of last cursor positions is stored, click the button again to move to previous positions. • **Navigate forward**: Allows you to move forward in the history of cursor positions within editor views. • **Navigate to**: Click to open the list of last visited editor views. If supported by the editor, position data (such as line and column in the declaration or implementation part of a logic editor) is provided. Select an entry to open the editor with the cursor being placed at the position that is indicated. Editors that do not support position data, open at a random cursor position.
Toolbar	Contains buttons to execute the available tools as defined in the **Tools > Customize** dialog box.
Multi-tabbed **Navigators**	The following **Navigators** are available as tabs where the different objects of a project are organized in a tree structure: • **Devices tree** • **Applications tree** • **Tools tree** • **Functional tree** For further information, refer to the chapter *Multi-Tabbed Navigators*, page 35.
Messages view	Provides messages on precompile, compile, build, download operations. Refer to the description of the **Messages** view commands for details (see Menu Commands, Online Help).
Information bar of the **Messages** view	Provides information messages, for example on the license status.
Status bar	Provides the following information: • Information about the user account. • Information about the editing mode and position if an editor is open. For further information, refer to Status Bar, page 33.
Multi-tabbed **Catalog** view	The **Catalog** view consists of different tabs: • **Hardware Catalog** ◦ **Controller** ◦ **HMI & iPC** ◦ **Devices & Modules** ◦ **Diverse** • **Software Catalog** ◦ **ToolBox** For further information, refer to the chapter *Multi-Tabbed Catalog Views*, page 45.

Component	Description
Status & Staging view	The **Status & Staging** view is used for version control support. For further information, refer to the **Status & Staging** view in the Menu Commands Online Help.
Multi-tabbed editor view	Used for creating the particular object in the respective editor. In the case of language editors (for example, ST editor, CFC editor), usually the window combines the language editor in the lower part and the declaration editor in the upper part. In the case of other editors, it can provide dialog boxes (for example, task editor, device editor). The name of the POU or the resource object is displayed in the title bar of this view. You can open the objects in the editor view in offline or online mode by executing the **Edit Object** command.

Status Bar

The bar at the lower border of the Logic Builder screen provides various types of information:

- Information on the logged-in user.
- If you are working in an editor view: the position of the cursor and the status of editing mode.
- In offline mode: the status of the program (such as compiler messages and precompile status). Double-click an icon to open the **Messages** view (see Menu Commands, Online Help) with the **Build** category selected.
- In online mode: the status of the program.
- Information on errors detected on EcoStruxure Machine Expert backend services.

Information on the logged-in user

Each project has a user and access management setting (refer to the **Project > User Management > Permissions...** command (see Menu Commands, Online Help)). The logged-in user name is displayed in the status bar.

Cursor positions in editor views

The cursor position is counted from the left or upper margin of the editor view.

Abbreviation	Description
Ln	Line in which the cursor is placed.
Col	Column in which the cursor is placed. (A column includes exactly one space, character, or digit.)
Ch	Number of characters. (In this context, a character can be a single character or digit as well as a tab including, for example, four columns.)

Double-click to open the dialog box **Go To Line**. Here you can enter a different position where the cursor is placed.

The status of the editing mode is indicated by the following abbreviations:

Abbreviation	Description
INS	Insert mode
OVR	Overwrite mode

Double-click this field to toggle the setting.

The status of the active application is indicated in offline mode by an icon and a tooltip:

Icon	Tooltips	Description
	**No reference code generated**	Application not downloaded.
	Code unchanged	Application not modified. A connection to the controller can be established without download.
	Online change possible	Application modified, can be downloaded by an online change.
	**Code download necessary**	Application modified, cannot be downloaded by an online change. A full download is required.

The following status of the program is indicated in online mode:

Text	Description
Program unchanged	Program on device matches the active application in the programming system.
Program modified (Online Change)	Program on device differs from the active application in the programming system, online change required.
Program modified (Full download)	Program on device differs from the active application in the programming system, full download required.

Online mode information

Status of the application on the device:

Text	Background Color	Description
RUN	Green	Program running.
STOP	Red	Program stopped.
HALT ON BP	Red	Program halted on a breakpoint.
The following status field is only available if the controller, depending on a setting in the device description, supports cycle-independent monitoring.		
IN CYCLE	White	Indicates that the values of the monitored expressions are read within one cycle.
OUT OF CYCLE	Red	Indicates that the retrieval of the values of the monitored variables cannot be performed within one cycle.

Operating mode information

The icon **Debug**, **Locked** or **Operational** indicates the present operating mode. For further information, refer to the chapter *Operating Modes* in the Menu Commands Online Help.

Security Information

The icon  **Security** opens the **Security Screen** editor (see Menu Commands, Online Help).

EcoStruxure Machine Expert backend services information

The **Services** icon is displayed in orange in the status bar when an error is detected on EcoStruxure Machine Expert backend services. In parallel, a **Service Status** pop-up message is displayed, listing the service or services that require attention. For further information on the detected error, click the **Services** icon or the **Open Service Health** link from the **Service Status** message to open the **Service Health** tab of the **Machine Expert Logic Builder** dialog box. For further information, refer to the Menu Commands Online Help.

Watch Windows and Online Views of Editors

Watch windows and online editor views display a monitoring view of a POU or a user-defined list of watch expressions.

Windows, Views, and Editors

There are two different types of windows in the Logic Builder:

- Some can be docked to any margin of the Logic Builder window or can be positioned on the screen as undocked windows independently from the Logic Builder window. Additionally they can be hidden by being represented as a tab in the Logic Builder window frame (refer to the *Customizing the User Interface* chapter, page 46). These windows display information which is not dependent on a single object of the project (for example **Messages** view or **Devices tree**). You can access them with the **View** menu (see Menu Commands, Online Help). Most views include a non-configurable toolbar with buttons for sorting, viewing, searching within the window.
- Other windows open when you are viewing or editing a specific project object in the respective editor. They are displayed in the multi-tabbed editor view. You cannot hide or undock them from the Logic Builder window. You can access them with the **Window** menu.

Switching Windows

You can switch between open views and editors by pressing the **Ctrl** and **Tab** keys simultaneously. A window opens that lists the views and editors that are open. As long as the **Ctrl** key is pressed the window stays open. Use the **Tab** key or the **Arrow** keys simultaneously to select a specific view or editor.

Multi-Tabbed Navigators

Overview

The multi-tabbed **Navigators** are default components of the Logic Builder screen.

By default, the following navigators are available:

- **Devices tree**: It allows you to manage the devices on which the application is to run.
- **Applications tree**: It allows you to manage project-specific as well as global POUs, and tasks in a single view.
- **Tools tree**: It allows you to manage project-specific as well as global libraries or other elements in a single view.
- **Functional tree**: It allows you to group the content of a controller according to your individual requirements.
- **Modules** view: It allows you to manage Smart Template modules. For detailed information, refer to the Smart Template Modules User Guide.

You can access views with the **View** menu.

Adding Elements to the Navigators

The root node of a navigator represents a programmable device. You can insert further elements below this root node.

To add elements to a node of a **Navigator**, select a device or object in the hardware or software catalog on the right-hand side of the Logic Builder screen and drag it to the **Navigator** (for example, the **Devices tree**). The node or nodes where the selected device or object fits are automatically expanded and displayed in bold. The other nodes where the selected device or object cannot be inserted are grayed. Drop the device or object on the suitable node and it is inserted automatically. If any further elements are required for the device or object, such as communication managers, they are inserted automatically.

Alternatively, you can select a node in the tree. If it is possible to add an object to the selected device or object, a green plus button is displayed. Click this plus button to open a menu providing the elements available for insertion.

It is also possible to add an object or a device, by right-clicking a node in a **Navigator** and running the command **Add Object** or **Add Device**. The device type which can be inserted depends on the selected object within the **Navigator**. For example, modules for a PROFIBUS DP slave cannot be inserted without having inserted an appropriate slave device before. Note that only devices correctly installed on the local system and matching the present position in the tree are available for insertion.

Repositioning Objects

To reposition objects, use the clipboard commands (**Cut, Copy, Paste, Delete**) from the **Edit** menu. Alternatively, you can drag the selected object with the mouse while the mouse-button (plus CTRL key for copying) is pressed. When you add devices using the copy and paste function, the new device gets the same name followed by an incrementing number.

Updating the Version of a Device

A device that is already inserted in the **Navigators** can be updated to another version or converted to another device.

Refer to the description of the separate commands:

- **Update Device** command, page 66
- **Convert Device** command, page 67

Description of the Devices Tree

Each device object in the **Devices tree** represents a specific (target) hardware object.

Examples: controller, fieldbus node, bus coupler, drive, I/O module

Devices and subdevices are managed in the **Devices tree**. Other objects which are needed to run an application on a controller are grouped in the other **Navigators**.

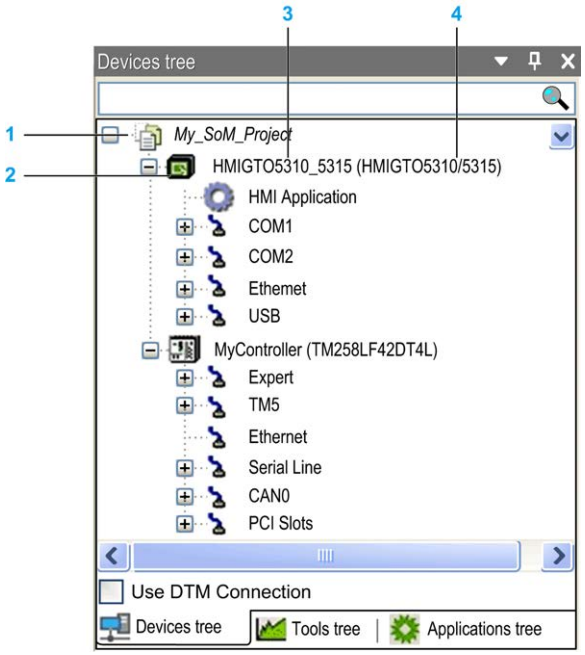
- The root node of the tree is a symbolic node entry: <projectname>
- The controller configuration is defined by the topological arrangement of the devices in the **Devices tree**. The configuration of the particular device or task parameters is performed in corresponding editor dialogs. Also refer to the chapter *Task Configuration*, page 233.

  Thus the hardware structure is mapped and represented within the **Devices tree** by the corresponding arrangement of device objects, allowing you to set up a complex heterogeneous system of networked controllers and underlying fieldbusses.
- To add devices configured with DTMs (Device Type Managers) to your project, activate the check box **Use DTM Connection** in the lower part of the **Devices tree**. This has the effect that a node **FdtConnections** is added below the root node of the tree. Below the **FdtConnections** node, a communication manager node is inserted automatically. You can add the suitable DTM device to this node.

  For further information, refer to the Device Type Manager (DTM) User Guide.
- To acknowledge diagnostic messages of an individual device with or without the subordinate devices, run the command **Acknowledge Diagnosis** or **Acknowledge Diagnosis for Subtrees** from the contextual menu of a device object.

  Device objects are marked when diagnostic messages are available:
  - A red exclamation mark indicates that the detected error is still valid.
  - A gray exclamation mark indicates that the detected error is no longer valid.

Example of a **Devices tree**:



1 Root node

2 Programmable device (with applications)

3 Symbolic device name

4 Device name defined in device description file

- Each entry in the **Devices tree** displays the symbol, the symbolic name (editable), and the device type (= device name as provided by the device description).
- The  symbol marks devices and tasks that are related to bus cycle settings and bus cycle tasks. For devices, a tooltip displays the related tasks. For tasks, a tooltip displays the related devices.
- A device is programmable or configurable. The type of the device determines the possible position within the tree and also which further resources can be inserted below the device.
- Within a single project, you can configure one or several programmable devices - regardless of manufacturer or type (multi-resource, multi-device, networking).
- Configure a device concerning communication, parameters, I/O mapping in the device dialog box (device editor). To open the device editor, double-click the device node in the **Devices tree** (refer to the description of the device editor, page 85).
- In online mode, the status of a device is indicated by an icon preceding the device entry:
  -  Controller is connected, application is running, device is in operation, data is exchanged. The option **Update IO while in stop** in the **PLC settings** view of the device editor, page 128 can be enabled or disabled.
  -  Controller is connected and stopped (STOP). The option **Update IO while in stop** in the **PLC settings** view of the device editor, page 128 is disabled.
  -  Device is not exchanging data, bus error detected, not configured or simulation mode (refer to the description of the **Simulation** command).
  -  Device is running in demo mode for 30 minutes. After this time, the demo mode expires and the fieldbus stops exchanging data.
  -  Device is configured but not fully operational. Data is not exchanged. For example, CANopen devices are in startup and preoperational.
  -  Redundancy mode active: The fieldbus master is not sending data because another master is in active mode.
  -  Device description was not found in device repository. For further information on installing and uninstalling devices in the **Device Repository** dialog box, refer to the description of the **Device Repository** (see Menu Commands, Online Help).
  -  The device is running, but a subordinate device is not running or a diagnostic message has been issued. The subordinate device is not visible because the **Devices tree** is collapsed.
  -  A diagnostic was pending. The issue that caused the error is no longer valid or has been solved. This symbol can be displayed in connection with various other symbols in this list.
  -  The device is not running or a diagnostic is pending. The issue that caused the error is still valid. This symbol can be displayed in connection with various other symbols in this list.
- The names of the connected devices and applications are displayed green shaded.
- The names of devices running in simulation mode (refer to the description of the **Simulation** command) are displayed in italics.
- Additional diagnostic information is provided in the **Status** view of the device editor, page 144.

You can also run the active application on a simulation device which is by default automatically available within the programming system. Therefore, no real target device is needed to test the online behavior of an application (at least that which does not rely on hardware resources for execution). When you switch to simulation mode (see Menu Commands, Online Help), an entry in the **Devices tree** is displayed in italics, and you can log into the application.

For information on the conversion of device references when opening projects, refer to the *Compatibility and Migration User Guide*.

Arranging and Configuring Objects in the Devices Tree

Adding devices / objects:

To add devices or objects to the **Devices tree**, select a device or object in the hardware catalog on the right-hand side of the Logic Builder screen and drag it to the **Devices tree**. The node or nodes where the selected device or object fits is expanded and is displayed in bold. The other nodes where the selected device or object cannot be inserted are grayed. Drop the device or object on the suitable node and it is inserted automatically.

Alternatively, you can select a node in the tree. If it is possible to add an object to the selected device or object, a green plus button is displayed. Click the plus button to open a menu providing the elements available for insertion.

Alternatively, you can add an object or a device, by right-clicking a node in the **Devices tree** and running the command **Add Object** or **Add Device**. The device type which can be inserted depends on the selected object within the **Devices tree**. For example, modules for a PROFIBUS DP slave cannot be inserted without having inserted an appropriate slave device before. No applications can be inserted below non-programmable devices.

Note that only devices correctly installed on the local system and matching the present position in the tree are available for insertion.

Repositioning objects:

To reposition objects, use the clipboard commands (**Cut, Copy, Paste, Delete**) from the **Edit** menu. Alternatively, you can draw the selected object with the mouse while the mouse-button (plus CTRL key for copying) is pressed. Consider for the **Paste** command: In case the object to be pasted can be inserted below or above the selected entry, the **Select Paste Position** dialog box opens. It allows you to define the insert position. When you add devices using the copy and paste function, the new device gets the same name followed by an incrementing number.

Updating the version of a device:

A device that is already inserted in the **Devices tree** can be replaced by another version of the same device type or by a device of another type (device update). In doing so, a configuration tree indented below the respective device is maintained as long as possible.

Adding devices to the root node:

Only devices can be positioned on the level directly below the root node <projectname>. If you choose another object type from the **Add Object** dialog box, such as a **Text list** object, it is added to the **Global** node of the **Applications tree**.

Subnodes:

A device is inserted as a node in the tree. If defined in the device description file, subnodes are inserted automatically. A subnode can be another programmable device.

Inserting devices below a device object:

You can insert further devices below a device object. If they are installed on the local system, they are available in the hardware catalog or in the **Add Object** or **Add Device** dialog box. The device objects are sorted within the tree from top to bottom: On a particular tree level first the programmable devices are arranged, followed by any further devices, sorted alphabetically.

Description of the Applications Tree

The **Application** objects, task configuration, and task objects are managed in the **Applications tree**.

The objects needed for programming the device (applications, text lists, etc.) are managed in the **Applications tree**. Devices that are not programmable (configuration only) cannot be assigned as programming objects. You can edit the values of the device parameters in the parameter dialog box of the device editor.

Programming objects, like particular POUs or global variable lists can be managed in 2 different ways in the **Applications tree**, depending on their declaration:

- When they are declared as a subnode of the **Global** node, these objects can be accessed by any devices.
- When they are declared as a subnode of the **Applications** node, these objects can only be accessed by the other devices declared in this **Applications** node.

You can insert an **Application** object only in the **Applications tree**.

Below each application, you can insert additional programming objects, such as **DUT, GVL**, or visualization objects. Insert a task configuration below an application. In this task configuration, the program calls have to be defined (instances of POUs from the **Global** node of the **Applications tree** or device-specific POUs). Note that the application is defined in the **I/O Mapping** view of the respective device editor, page 146.

The POUs are displayed in different colors that have the following meaning:

Color	Description
Black	Default color.
Gray	Displayed after a code generation. The POU is not used in the project
Blue	Displayed after a code generation and when the project has already been downloaded. The POU has changed as compared to the POU on the controller and will be included with the next download.
Blue-green	The **Exclude from build** option is selected for the POU in the **Build** tab of the **Properties** dialog box.

Comparing Objects

By right-clicking an object in the **Applications tree**, the following commands are provided in the contextual menu for comparison.

Command	Description
Compare both selected objects...	This command is available if two objects are selected in a navigator. Execute the command to compare the two objects. If two objects of different types are selected, a corresponding message is displayed.
Select first object for compare...	The selected object is used as basis for the comparison.
Compare with selected object...	This command is available if a second object is selected that is of the same type as the object that has previously been selected with **Select first object for compare...**. Execute the command to compare the two objects. The result is displayed in a comparison view similar to the **Project Comparison - Differences** view (for further information, refer to the Menu Commands online help).
Compare with Selected (Raw)	This command is exclusive to projects stored in file-based storage format. It is available if an object has previously been selected with **Select first object for compare...** and a second object is selected. The second object can have a different type than the first object. This process compares the raw contents of the objects and allows you to compare, for example, the content of two **Action** objects that belong to two different function blocks. The result is displayed in a comparison view similar to the **Project Comparison - Differences** view (for further information, refer to the Menu Commands online help).
Compare with device...	Execute the command to compare the open project with the project that is running on a configured controller. For detailed information, refer to **Compare with device...** in the Menu Commands online help.
Compare with target device...	Execute the command to compare the loaded project with the project that is running on a configured controller, whereas configured means that you must have downloaded the source code to the controller beforehand. For detailed information, refer to **Compare with target device...** in the Menu Commands online help.

Description of the Tools Tree

The **Tools tree** contains objects, that are supplemental to the application, as, for example, libraries, visualizations, recipe manager, alarm configuration. Devices that are only configured (not programmed) cannot be assigned the programming objects described above. You can edit the values of the device parameters in the parameter dialog box of the device editor.

Programming objects, like the **Library Manager**,can be managed in 2 different ways in the **Tools tree**, depending on their declaration:

- When they are declared as a subnode of the **Global** node; then these objects can be accessed by any devices.
- When they are declared as a subnode of the **Applications** node; then these objects can only be accessed by the other devices declared in this **Applications** node.

Functional tree

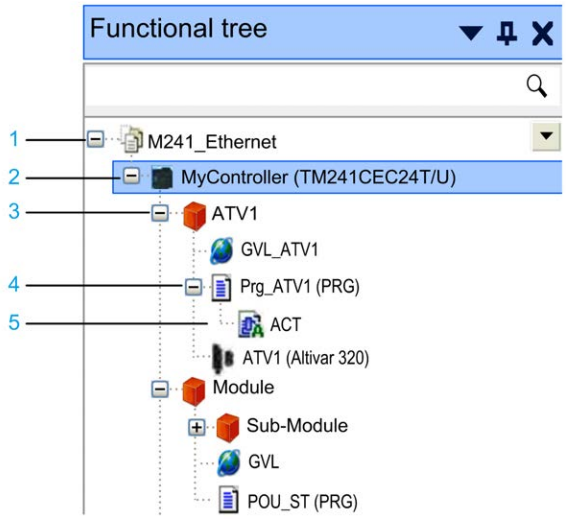
Overview

The **Functional tree** is available for controllers that have a **Functional Model** node in the **Devices tree**. It allows you to group multiple objects, such as IEC

code or devices, and link them to a function. Once this function is created, you can reuse it. By creating this modularity, reuse your developments to improve your vision of the project. You can export / import the **Functional tree** and reuse it in another project.

Description of the Functional tree

Example of a **Functional tree**:



1 Root node: corresponds to the name of the open project

2 Controller node: only those controllers that have a **Functional Model** node in the **Devices tree** are displayed

3 Functional module: nodes for structuring the **Functional tree**

4 Attached object: object attached to the functional module

5 Child object: child object of the attached object

Selecting Controllers

Select controllers for the **Functional tree** as follows:

Step	Action	Result
1	In the **Functional tree**, right-click the root node, and run the command **Select Controllers**.	A new subnode **Functional Model** is inserted for each selected controller in the **Devices tree**.
2	In the **Select Controllers** dialog box, select the controllers you want to add to the **Functional tree**, and click **OK**.	New controller nodes are added to the **Functional tree** below the root node for each selected controller.

Adding Nodes

To group the content of a controller according to your individual requirements, the **Functional tree** allows you to create subnodes below the controller nodes.

Node	Description	How to create
Functional module	A functional module is a group of program elements intended to perform an application function. Functional module nodes create a hierarchical structure in the **Functional tree**. To create a meaningful structure, edit the default name and assign a name of your choice to each functional module.	Select a parent node (for example, the controller node), and click the green plus button.
Attached object	Attached objects are nodes of the other navigators (**Devices tree**, **Applications tree**, **Tools tree**) that represent the content of the controller. Note the following: • One object can only be attached to one functional module. • An object can only be attached to a function module of the same controller. • You can attach only those objects that are also allowed in function templates, page 782.	Right-click a functional module node, and run the command **Select Objects** from the contextual menu. From the **Select Objects** dialog box, select the node you want to attach and click **OK**.
Child object	Child objects of the attached objects.	Child objects are displayed in the **Functional tree**.

Deleting Nodes

To delete a node from the **Functional tree**, right-click on it, and run the command **Delete** from the contextual menu. You are requested to delete the selected object, with its child objects, only from the **Functional tree** or from the whole project.

Child objects cannot be removed from the **Functional tree** only. If you intend to delete a child object, you are prompted to confirm that the object is removed from the whole project.

Reusing Functional Modules

If you have created a functional module that you want to reuse in the same or in another project, use the function templates as they can resolve the dependencies between the attached objects. The **Import**/**Export** commands and the copy/paste functions can also be used, but they only serve special cases as described in the following sections.

Reusing Functional Modules by Using Function Templates

You can save a functional module to a function template by right-clicking the node and run the command **Save as Function Template** from the contextual menu.

To instantiate a functional module from a function template, right-click a node in the **Functional tree**, and run the command **Add Function From Template** from the contextual menu.

For further information, refer to the *Managing Function Templates* chapter, page 779.

Reusing Functional Modules by Using the Import/Export Commands

When you use the **Project > Export** command and the **Project > Import** command (see Menu Commands, Online Help) for reusing functional modules, note the following:

If...	Then ...
If you export a complete controller device, and then import it in the same or in another project,	Then the functional model is recreated.
If you export and import the functional model only,	Then the attached objects are not recreated.

Reusing Functional Modules by Using the Copy and Paste Functions

When you use the copy and paste functions for reusing functional modules, note the following:

If...	Then ...
If you copy a complete controller device, and then paste it in the same or in another project,	Then the functional model is recreated.
If you copy and paste one or more functional modules only,	Then the attached objects are not recreated.

It is not possible to copy and paste attached objects in the **Functional tree**.

Multi-Tabbed Catalog View

Overview

The multi-tabbed **Hardware Catalog** is a default component of the Logic Builder screen.

It contains the following tabs:

- **Controller**: Contains the **Logic**, **HMI**, and **Motion** controllers that can be inserted in your project.
- **Devices & Modules**: Contains the **PLC Components**, **I/O Modules**, and the **Communication**, **Motor Control**, **Safety**, and **Sensor** devices that can be inserted in your project. It also allows you to insert devices by using a device template.
- **HMI & iPC**: Contains the **HMI** and **iPC** devices that can be inserted in your project.
- **Diverse**: Contains third-party devices that can be inserted in your project.

The content of the individual tabs depends on the project. If the controllers integrated in the project do not support, for example, CANopen, then CANopen devices are not displayed in the catalogs.

You can extend this view by the **Software Catalog > ToolBox** via the menu **View > Software Catalog**.

The buttons **Hardware Catalog**  and **Software Catalog**  in the toolbar allow you to display or hide the catalog views.

You can add the elements from the catalogs to the project by drag and drop as described in the *Adding Devices by Drag and Drop* chapter, page 58.

Searching Within Catalogs

Each tab of the catalog view contains a search box. The sublists of the tab are analyzed for the string you enter in the search box. In open sublists, the identified entries are marked yellow. Any other items of the list that do not correspond to the search string are hidden. The number of items found in closed sublists is displayed in bold in the title bar of each sublist.

By default, the search is executed on the names of the items in the lists. But the tagging mechanism is also supported. It allows you to assign search strings of your choice to any item included in the **Catalog** view.

Favorites List

Each tab of the catalog view contains a **Favorites** list. To provide quick access, you can add frequently used elements to this **Favorites** list by drag and drop.

Adding Devices From Device Templates in the Devices & Modules Tab

The **Devices & Modules** tab contains the option **Device Template** at the bottom. Activate this option to display the available templates of field devices in the lists of the **Devices & Modules** tab. Add them to the **Devices tree** as described in the *Adding Devices from Template* chapter, page 770.

Status & Staging View (exclusive to *.fbs projects)

This function is exclusive to projects stored in file-based storage format.

The **Status & Staging** view is provided for version control support.

For further information, refer to the **Status & Staging** view in the Menu Commands Online Help.

Customizing the User Interface

Overview

The look of the user interface, in terms of arrangement and configuration of the particular components, depends on the following:

- Default pre-settings for menus, keyboard functions, and toolbars. You can overwrite the default settings by using the **Customize** dialog box (see Menu Commands, Online Help) (by default available in the **Tools** menu). The present settings are saved on the local system. A reset function is available for restoring the default values at any time.
- Properties of an editor as defined in the respective **Tools > Options** dialog box (see Menu Commands, Online Help). You can also overwrite these settings. The present configuration is saved on the local system.
- The way you arrange views or editor windows within the project. The present positions are saved with the project (see below).
- The selected perspective. By default, the **Logic Configuration** perspective is selected. For further information, refer to the *Perspectives* paragraph in this chapter, page 49.

Arranging Menu Bars and Toolbars

The menu bar is positioned at the top of the user interface, between the window title bar and view windows. You can position a toolbar within the same area as the menu bar (fix) or as an independent window anywhere on the screen.

In view windows, such as the **Devices tree**, a special toolbar is available. It provides elements for sorting, viewing, and searching within the window. You cannot configure this toolbar.

Arranging Windows and Views

Closing a view or editor window: Click the cross button in the upper right corner.

Opening a closed view: you can reopen the views of default components with the **View** menu. To open an editor window, run the command **Project > Edit object** or double-click an entry in the **Devices tree**, **Applications tree**, or in the **Tools tree**.

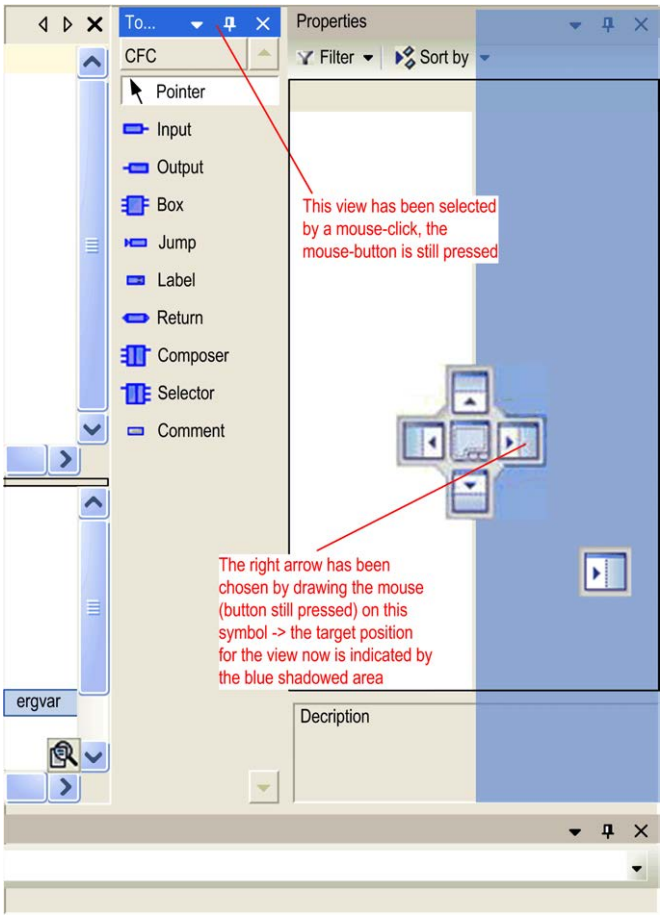
Resizing a view or window within the frame window: Move the separator lines between neighboring views. You can resize independent view windows on the desktop by moving the window borders.

Moving a view to another position on your desk top or within the frame window: Click the title bar or, in the case of tabbed views, click the tab of the view, keep the mouse-button pressed, and move the view to the desired place. Arrow symbols indicate possible target positions. The target position is indicated by a blue-shadowed area.

Arrow symbols indicating new position

Arrow symbol	Description
	View is placed above.
	View is placed below.
	View is placed to the right.
	View is placed to the left.
	View is placed here: the view currently placed at this position and the new one are arranged as icons.

Example of navigation by the arrow symbols



When you release the mouse-button, the view is placed at the new position.

Views with an **Auto Hide** button can be placed as independent windows (floating) anywhere on the screen by moving them and not dragging them on one of the arrow symbols. In this case, the view looses the **Auto Hide** button. As an alternative, run the commands **Dock** and **Float** from the **Window** menu.

Hiding views: You can hide views with **Auto Hide** buttons at the border of the window. Click the **Auto Hide** down button in the upper right corner of the view. The view will be displayed as a tab at the nearest border of the frame window. The content of the view is only visible as long as the cursor is moved on this tab. The tab displays the icon and the name of the view. This state of the view is indicated by the docking button changed to **Auto Hide**.

Unhiding views: To unhide a view, click the **Auto Hide** button.

An alternative way of hiding and unhiding a view is provided by the **Auto Hide** command that is by default available in the **Window** menu.

It is not possible to reposition the information and status bar on the lower border of the user interface, page 30.

Perspectives

A perspective is used to save the layout of views. It stores whether the **Messages** and **Watch** views are open and at which position the view windows are located (docked or independent windows).

By default, the following perspectives for specific use cases in the **Window > Switch Perspective** menu or in the perspective table in the toolbar are provided.

Perspective name	Use case	Navigators (on the left side)	Catalog views (on the right side)	Views at the bottom of the screen
Classic	Default views.	• **Devices** • **POUs**	–	• **Messages** (in **Auto Hide** mode) • **Watch 1** • **CallStack**
ETest	For working within the ETEST framework.	• **Devices** • **POUs**	–	• **Messages** • **Watch 1** • **CallStack** • **Testruns**
Device Configuration	For adding / configuring devices.	• **Devices tree**	Hardware catalog • **Controller** • **Devices & Modules** • **HMI & iPC** • **Diverse**	**Messages** (in **Auto Hide** mode)
Online	For online mode.	• **Devices tree** • **Applications tree** • **Tools tree**	–	• **Messages** (in **Auto Hide** mode) • **Watch 1**
Smart Template	For working with **Smart Template** modules.	• **Devices** • **POUs** • **Modules**	–	**Messages** (in **Auto Hide** mode)
Logic Configuration	For adding / creating logic.	• **Devices tree** • **Applications tree** • **Tools tree**	Hardware catalog • **Controller** • **Devices & Modules** • **HMI & iPC** • **Diverse**	**Messages** (in **Auto Hide** mode)

The **Online** perspective is automatically selected when the application is switched to online mode.

Creating your own perspective:

In addition to these default perspectives, you can create your own view layout and save it in different perspectives according to your individual requirements.

To create your own perspective, proceed as follows:

Step	Action
1	Resize, open, or close views according to your individual requirements.
2	Run the command **Save Perspective** from the **Window** menu to save your modifications to a new perspective.
3	In the **Save Perspective** dialog box, enter a name for your perspective. **Result**: The present view layout is saved. The new perspective is available in the **Window > Switch Perspective** menu and in the perspective table in the toolbar.

Resetting a perspective to its initial state:

To reset a modified perspective to its initial state, run the command **Reset current Perspective** from the **Window** menu.

Importing / exporting perspectives:

To be able to exchange perspectives between different installations or between different users, the **Tools > Options > Perspectives** dialog box (see Menu Commands, Online Help) allows you to export perspectives to an XML file and to import already available perspective XML files.

Zoom

Each editor window provides a zoom function. Click the zoom button  in the lower right corner of the window to open a list. It allows you to choose one of the zoom levels 25, 50, 100, 150, 200, and 400 percent or to enter a zoom factor of your choice. A printout always refers to the 100% view.

Customization of the user interface is possible in offline and in online mode.

User Interface in Online Mode

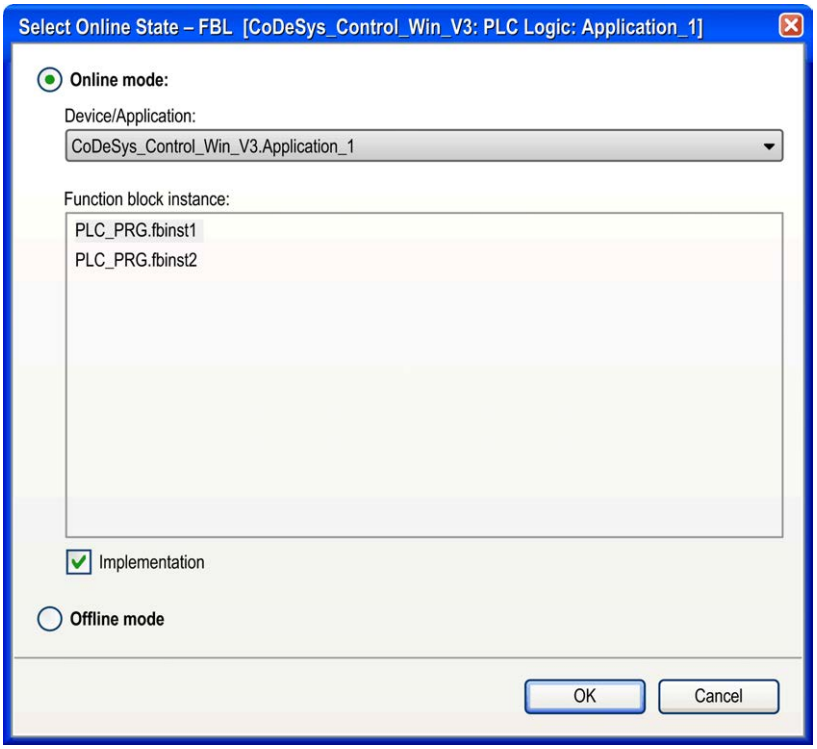
Overview

As soon as you log in with the project, the objects which have already been opened in offline mode are automatically viewed in online mode. The perspective is automatically switched to the **Online** perspective, page 49 which means that the **Watch** view opens by default.

To open an object in online mode, double-click the node in the **Applications tree** or execute the **Project > Edit Object** command.

If there are several instances of the selected object (such as function blocks) contained in the project, a dialog box named **Select Online State <object name>** is displayed. It allows you to choose whether an instance or the base implementation of the object should be viewed and whether the object should be displayed in online or offline mode.

Select Online State dialog box



The **Device/Application** field contains the device and application to which the object is associated.

To open the online view of the object, activate the option **Online mode** and click **OK**. To see the offline view, activate the option **Offline mode**.

If the object is a function block, the **Function block instance** field contains a list of the instances currently used in the application.

In this case, the options available are:

- Either select one of the instances and activate **Online** or **Offline mode**.
- Or select the option **Implementation** which - independently of the selected instance - will open the base implementation view of the function block. The **Implementation** option has no affect for non-instantiated objects.

For more information on the online views of the particular editors, refer to the respective editor descriptions.

The status bar, page 30 provides information on the current status of the application.

Menus and Commands

Overview

The following figure shows the default menu bar:



Some commands are not visible in the default view. To add a command to a menu or reorganize the menu structures, use the **Tools > Customize** dialog box.

Specific commands, for a particular editor for example, are usually available in a corresponding menu. These commands are only visible when the editor is open. For example: when you edit an object in the SFC editor, the **SFC** menu is added to the menu bar.

For a description of the menus and commands, refer to the separate Menu Commands Online Help (see Menu Commands, Online Help).

Basic Concepts

What's in This Chapter

Introduction and Basic Concepts 53

Introduction and Basic Concepts

Overview

EcoStruxure Machine Expert is a device-independent controller programming system.

Conforming to the IEC 61131-3 standard, it supports standard programming languages.

Object Orientation

The object-oriented approach is not only reflected by the availability of corresponding programming elements and features, but also in the structure and version handling and in the project organization. Multi-device usage of a project is possible based on jointly used, instantiated programming units.

Version Handling

A parallel installation of several versions of components and working with the desired combination of versions is possible. This also pertains to the device-specific use of different compiler versions. Individual functions can be added without having to update the whole version.

For further information, refer to the *Compatibility and Migration User Guide*.

Project Organization

Project organization is also determined in an object oriented fashion. A project contains a controller program composed of various programming objects and it contains definitions of the resources which are needed to run instances of the program (application) on defined target systems (devices, controllers).

So there are two main types of objects in a project:

Object Type	Description
Programming objects (POUs), page 173	These are programs, functions, function blocks, methods, interfaces, actions, data type, definitions, and so on.
Resource objects (**Devices tree**)	Device objects are only managed in the **Devices tree**. To insert objects in the **Devices tree**, refer to *Adding Elements to the Navigators* section, page 40.

Code Generation

Code generation by integrated compilers and the subsequent use of the resulting machine code provides for short execution times.

Data Transfer to the Controller Device

The data transfer between EcoStruxure Machine Expert and the device is conducted via a gateway (component) to a runtime system. After downloading the application to the controller it can be monitored and controlled in EcoStruxure Machine Expert.

Supported Programming Languages

The programming languages mentioned in the standard IEC 61131 are supported by specially adapted editors:

- FBD/LD/IL editor, page 268 for function block diagram (FBD), ladder logic diagram (LD), and instruction list (IL)
- Ladder editor, page 303 for ladder logic diagram (LD)
- SFC editor, page 330 for sequential function chart
- ST editor, page 356 for structured text

Additionally, an editor for programming in CFC is provided that is not part of the IEC standard:

- CFC editor, page 310 for continuous function chart

CFC is an extension to the standard IEC programming languages.

A CFC - page-oriented editor is also provided. It allows you to arrange the CFC elements on individual pages.

Configuration

What’s in This Part

Installing Devices 56
Managing Devices 58
Common Device Editor Dialogs 85

Installing Devices

What's in This Chapter

Integration of Sercos Devices from Third-Party Vendors 56

Integration of Sercos Devices from Third-Party Vendors

Introduction

Use the **Device Repository** dialog box (see Menu Commands, Online Help) to integrate Sercos devices with generic I/O profiles in your programming system.

To install this Sercos device, you need the SDDML (Sercos Device Description Markup Language) file (device description file for Sercos devices) provided by the vendor of the device. The SDDML file is a device description file for Sercos devices.

There are two types of Sercos devices with generic I/O profiles available:

- Block I/O devices

  A block I/O device is a pre-assembled block that consists of a bus interface and an I/O module.
- Modular I/O devices

  Modular I/O devices are I/O modules which can be connected to a bus interface.

Integrating In the Logic Builder

Proceed as follows to integrate Sercos devices from third-party vendors in your programming system:

Step	Action
1	Select **Tools > Device Repository...** from the menu bar. **Result**: The **Device Repository** dialog box opens.
2	Click the **Install...** button in the **Device Repository** dialog box. **Result**: The **Install Device Description** dialog box opens.
3	Select the file type **SERCOS III I/O device descriptions (*.xml)** and browse your file system for the SDDML file to open.
4	Select the SDDML file and click **Open**. **Result**: The SDDML file is converted and imported into a compatible file format.

NOTE: If the selected SDDML file is not compatible or if the Sercos device of the third-party vendor is not using a compatible FSP (Function Specific Profile) type, then a corresponding diagnostic message is indicated in the **Messages** view (see Menu Commands, Online Help).

Verifying the Integration

To verify whether a Sercos device with generic I/O profile has been integrated in your programming system, proceed as follows:

Step	Action
1	Select **Tools > Device Repository...** from the menu bar. **Result**: The **Device Repository** dialog box opens.
2	In the tree structure **Installed device descriptions**, expand the node **Fieldbusses > Sercos**.
3	Expand the subnode **Slave** to verify whether the Sercos bus interfaces that you integrated are available in the list.
4	Expand the subnode **Module** to verify whether the Sercos I/O modules that you integrated are available in the list.

For further information, refer to the description of the **Device Repository** dialog box (see Menu Commands, Online Help).

Managing Devices

What's in This Chapter

Adding Devices through the **Hardware Catalog** 58
Adding Devices by Contextual Menu or Plus Button 60
Updating Devices 66
Converting Devices 67
Converting Projects 69

Adding Devices through the Hardware Catalog

Adding Devices by Drag and Drop

Overview

A multi-tabbed catalog view is provided on the right-hand side of the Logic Builder.

To add a device to the **Devices tree**, select the respective entry in the **Hardware Catalog**, drag it to the **Devices tree**, and drop it at a suitable node. It is added automatically to your project.

Adding Controllers

To add a controller to your project, proceed as follows:

Step	Action
1	Open the **Hardware Catalog** by clicking the **Hardware Catalog** button  in the toolbar if it is not already opened.
2	Select the tab **Controller** in the **Hardware Catalog**. **Result**: The controllers suitable for your project are displayed in the **Hardware Catalog**.
3	Select a controller entry in the **Controller** tab, drag it to the **Devices tree** and drop it at a suitable node. You can drop a controller at any empty space inside the **Devices tree**. **Result:** The controller is added to the **Devices tree** as a new node with different subnodes depending on the controller type.

Adding Expansion Devices

To add an expansion device to a controller, proceed as follows:

Step	Action
1	Open the **Hardware Catalog** by clicking the **Hardware Catalog** button  in the toolbar if it is not already opened.
2	Select the tab **Devices & Modules** in the **Hardware Catalog**. **Result**: The expansion devices suitable for your project are displayed in the **Hardware Catalog**.

Step	Action
3	Select your expansion device, drag it to the **Devices tree** and drop it at a suitable subnode of a controller. **NOTE:** Suitable subnodes are expanded and highlighted. **Result:** The expansion device is added to the **Devices tree** below the subnode of the controller.
4	If the expansion device requires a communication manager, this node is added automatically to the **Devices tree**. If several communication managers are available for your expansion device, a dialog box is displayed allowing you to select the suitable communication manager.

Adding Devices and Modules

To add a field device to a controller, proceed as follows:

Step	Action
1	Open the **Hardware Catalog** by clicking the **Hardware Catalog** button  in the toolbar if it is not already opened.
2	Select the tab **Devices & Modules** in the **Hardware Catalog**. **Result**: The field devices suitable for your project are displayed in the **Hardware Catalog**.
3	Select a field device entry in the **Devices & Modules** catalog view, drag it to the **Devices tree**, and drop it at a suitable subnode of a controller. **NOTE:** Suitable subnodes are expanded and highlighted. **Result:** The field device is added to the **Devices tree** below the subnode of the controller.
4	If the field device requires a communication manager, this node is added automatically to the **Devices tree**. If several communication managers are available for your field device, a dialog box is displayed allowing you to select the suitable communication manager.

Adding Devices from Device Template

To add a device from a device template, proceed as follows:

Step	Action
1	Open the **Hardware Catalog** by clicking the **Hardware Catalog** button  in the toolbar if it is not already opened.
2	Select the tab **Devices & Modules** in the **Hardware Catalog**.
3	Select the option **Device Template** at the bottom of the **Devices & Modules** tab. **Result**: The device templates suitable for your project are displayed in the **Devices & Modules** tab.
4	Add them to the **Devices tree** as described in the *Adding Devices from Template* chapter, page 770.

Adding Devices by Contextual Menu or Plus Button

Adding a Controller

Introduction

To add devices to the **Devices tree**, you can also click the green plus button that is displayed at the suitable node in the **Tree**. Alternatively, you can right-click a node of the **Tree** to use the contextual menu. The **Add Device** dialog box opens that allows you to determine whether the device will be appended, inserted, or plugged to the selected node (see Menu Commands, Online Help).

When you add a controller to your project, several nodes are automatically added to the **Devices tree**. These subnodes are controller-specific, depending on the functions the controller provides.

The following paragraph describes the general procedure of adding a controller. For details on a specific controller, refer to the programming manual for your particular controller.

Adding a Controller

To add a controller to your project, proceed as follows:

Step	Action
1	Select a project node, right-click the project node and run the **Add Device...** command from the contextual menu. **Result:** The **Add Device** dialog box opens.
2	In the **Add Device** dialog box, select **Schneider Electric** from the list box **Vendor**.
3	Choose the controller you want to insert into the project.
4	Rename your device by typing a name in the text box **Name**. **NOTE:** Choose a name that complies to the IEC standard. Do not use special characters, leading digits, or spaces within the name. The name must not exceed a length of 32 characters. If you do not rename the device, a name is given by default. Naming the device meaningfully may ease the organization of your project.
5	Click the **Add Device** button. **Result:** The selected controller is added to the project and appears as a new node in the **Devices tree**. The **Add Device** dialog box remains open. You can do the following: • You can add another controller by going back to step 3 • You can click the **Close** button to close the **Add Device** dialog box

Adding Expansion Devices

Available Expansion Devices

For a list of expansion devices available for the different controllers, refer to the *Supported Devices* chapter of the *Introduction* document.

⚠WARNING

UNINTENDED EQUIPMENT OPERATION

- Only use software approved by Schneider Electric for use with this equipment.
- Update your application program every time you change the physical hardware configuration.

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

Adding Expansion Devices

To add expansion devices to your device, proceed as follows:

Step	Action
1	Select a controller node and click the green plus button of the node or right-click the controller node and execute the **Add Device...** command from the contextual menu. **Result:** The **Add Device** dialog box opens.
2	In the **Add Device** dialog box, select **Schneider Electric** from the **Vendor** list.
3	Choose the expansion device you want to add to your controller from the **Device** list below.
4	Rename your expansion device by typing a name in the text box **Name**. **NOTE:** The name must not contain any space character. If you do not rename the expansion device, a name is given by default. Naming the expansion device meaningfully may ease the organization of your project.
5	Click the **Add Device** button. **Result**: The selected expansion device is added to the project and is displayed in the **Devices Tree** as a new subnode of your controller. The **Add Device** dialog box remains open. You can do the following: • You can add another expansion device by going back to step 3 of this description • You can click the **Close** button

Expansion Device Configuration

For more information about configuration, refer to the *Programming Guide* of your expansion device.

Adding Communication Managers

Overview

Communication managers are mandatory to activate and configure any hardware bus interface, for example CANopen or serial line.

Two types of communication managers exist:

- Fieldbus managers which allow to configure fieldbus devices (for example CANopen or Modbus devices)
- General communication managers

Available communication managers are listed below:

Name	Interface type	Description
ASCII Manager	Serial line	Used to transmit and/or receive data with a simple device.
Machine Expert-Network Manager	• Serial line (maximum 1) • Ethernet (maximum 3)	For exchanging data with HMIs.
Modbus IOScanner	Serial line	Modbus RTU or ASCII protocol manager used to define implicit exchanges (I/O scanning) with Modbus devices.
Modbus Manager	Serial line	Used for Modbus RTU or ASCII protocol in client or server mode.
CANopen Performance	CAN	CANopen manager for performance controllers (M241, M251).
CANopen_Manager	CAN	CANopen manager for PacDrive controllers.
Modbus TCP Slave Device	Ethernet	Modbus TCP manager for controllers with Ethernet port.
EthernetIP	Ethernet	EtherNet/IP manager for controllers with Ethernet port (M251).
Industrial Ethernet Network	Ethernet	Used to configure EtherNet/IP and Modbus TCP scanner services on M241 / M251 controllers with Ethernet port.
EtherNet/IP Scanner	Ethernet	Used to configure EtherNet/IP Scanner on M262 controllers.
ModbusTCP IOScanner	Ethernet	Used to configure Modbus TCP scanner services on M262 controllers.
Sercos Master	Ethernet	Used to configure a Sercos Master on M262 and PacDrive controllers.
PROFINETIO-Controller	-	Used to configure a PROFINETIO-Controller based on Hilscher netX on PacDrive controllers.
EtherNet-IP-Scanner	-	Used to configure an EtherNet-IP-Scanner based on Hilscher netX on PacDrive controllers.
TCP/UDP Manager	-	Used to create a customized protocol based on TCP or UDP for exchanging data with a third party device on M262 controllers.
PROFIBUS-DPV1-Master	-	Used to configure a ProfiBus master based on Hilscher netX on PacDrive controllers.
EtherCAT-Master	-	For configuring an EtherCAT master based on Hilscher netX on PacDrive controllers.

Adding the Communication Manager

Communication managers are automatically added with the respective device.

To add a communication manager separately, proceed as follows:

Step	Action
1	In the **Devices Tree**, select the bus interface (**Serial Line**, **CANopen bus** / **CANbus**, **Ethernet**) and click the green plus button of the node or right-click the bus interface node and execute the **Add Device...** command from the contextual menu. **Result**: The **Add Device** dialog box opens.
2	In the **Add Device** dialog box, select **Schneider Electric** from the list **Vendor**. **Note**: You can filter the devices by brand by clicking the list **Vendor**.
3	Select the **Communication manager** from the list.
4	Rename your device by typing a name in the **Name** box. **Note**: Do not use spaces within the name. If you do not rename the device, a name is given by default. Naming the device meaningfully may ease the organization of your project.
5	Click the **Add Device** button.
6	Click the **Close** button to close the **Add Device** dialog box.
7	Configure the **Communication manager**.

Adding Devices to a Communication Manager

Overview

You can add field devices to the communication manager by selecting the field device manager node (for example, CANopen or Modbus manager) in the **Devices Tree** and clicking the green plus sign. Alternatively, you can right-click the field device manager node in the **Devices Tree** and run the **Add Device** command.

As a prerequisite, the device must be available in the **Device Repository** dialog box (see Menu Commands, Online Help).

Adding Devices

Step	Action
1	Select the field device manager node (CANopen or Modbus manager) in the **Devices Tree** and click the green plus sign, or right-click the field device manager node and run the **Add Device...** command from the contextual menu. **Result**: The **Add Device** dialog box opens.
2	In the **Add Device** dialog box, select **Schneider Electric** from the list box **Vendor**. **Note**: You can filter the devices by brand by clicking the list box **Vendor**.
3	Select the device of your choice from the list below.
4	Rename your device by typing a name in the **Name** textbox. **NOTE:** Do not use spaces within the name. Do not use an underscore character (_) at the end of the name. If you do not rename the device, a name is given by default. Naming the device meaningfully may ease the organization of your project.
5	Click the **Add Device** button. **Result**: The field device is added to the field device manager. **NOTE:** The **Add Device** dialog box remains open. You can do the following: • You can add another device by going back to step 2. • You can click the **Close** button.

Access to Diagnostic Information

To get diagnostic information of devices on CANopen, use the CAA_CiA405. library.

Access to Configuration Diagnostic (for Advanced Users)

You can use the options **Abort if error** and **Jump to line if error** in the **Service Data Object** tab of the CANopen configurator to manage potential configuration inconsistencies.

To optimize the CAN master performance, CAN diagnostics are external from the CAN master in the controller. The CAN diagnostic structure is defined in the CanConfig Extern library available in the **Library Manager**.

The structure `g_aNetDiagnosis` contains the most recent diagnostic information from the slaves. The structure is updated every time a slave is configured.

This structure can be used within the program to do the following:

- Monitoring the response of the slaves configured by using SDO messages.
- Monitoring the master for abort messages from the slaves before allowing a machine / application start-up.

This structure must be in place and active within the user application during testing, debugging and commissioning of the application. When the machine and its controlling application have been commissioned and validated, then it would be possible to disable this code from execution to reduce traffic on the CANopen network.

However, if during the lifecycle of an application and the machine or process that it controls, slaves are added or replaced in the operational system, then the diagnostic structure should continue to remain active in the application.

⚠WARNING

UNINTENDED EQUIPMENT OPERATION

- Use the g_aNetDiagnosis data structure within the application to monitor CAN slave responses to configuration commands.
- Verify that the application does not start up or put the machine or process in an operational state in the event of receiving SDO abort messages from any of the CAN slaves.

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

After adding the CanConfig Extern library to your application, use the **Net Diagnostic** definition within your application to test for SDO abort messages from the CAN slaves.

The following code example illustrates the use of the CAN diagnostic data structure:

```
IF g_aNetDiagnosis[CAN_Net_Number].ctSDOErrorCounter = 0
THEN
    (* No error is detected in the configuration*)
ELSE
    (* An error has been detected during configuration. Get
the latest error information.*)
    // node ID of the slave which sent the abort code
    ReadLastErrorNodeID := g_aNetDiagnosis[CAN_Net_Number].
usiNodeID;
    // index used in the aborted SDO
    ReadLastErrorIndex := g_aNetDiagnosis[CAN_Net_Number].
wIndex;
    // subIndex used in the aborted SDO
    ReadLastErrorSubIndex := g_aNetDiagnosis[CAN_Net_
Number].bySubIndex;
    //SDO abort code
    ReadLastErrorSdoAbortCode := g_aNetDiagnosis [CAN_Net_
Number].udiAbortCode;
    (* Do not allow the start-up or other operation of the
machine or process *)
END_IF
```

NOTE: In this example, the `CAN_Net_Number` would be 0 for the CAN0 port and, if the controller is so equipped, 1 for the CAN1port.

Adding Devices from Template

Overview

It is also possible to add a new device using a device template. For a description of this procedure, refer to the *Managing Device Templates* section, page 770.

Updating Devices

Updating Devices

Introduction

The update device function allows you to update a device selected in the **Devices tree** by another version of the same device.

Updating Devices

To update a device of your project by another version or by a different device, proceed as follows:

Step	Action
1	Select the device you want to update in the **Devices tree** and execute the command **Update Device...** from the **Project** menu. OR Right-click the device you want to update in the **Devices tree** and execute the command **Update Device...** from the contextual menu. **Result**: The **Update Device** dialog box opens. OR Right-click the device you want to update in the **Devices tree** and execute the command **Add Device...** from the contextual menu. In the **Add Device** dialog box select the **Action: Update device**. **Result**: The **Add Device** dialog box is converted into the **Update Device** dialog box.
2	From the **Device:** list, choose the version that should replace the present device. To select a specific version of the device, select the options **Display all versions (for experts only)** and/or **Display outdated versions**.
3	Click the **Update Device** button. **Result:** The device that had been selected in the **Devices tree** is updated by the new device version. The new device version is now displayed at the selected node in the **Devices tree**.

Effects after Updating a Device

The subdevices that are located in the **Devices tree** below the device you updated are automatically updated as well.

The device configuration settings are not modified if the device type has not been changed.

If the update procedure causes any mismatch in the existing configuration, this is detected at the next **Build** run of the application. Detected mismatches are indicated by a message. This also concerns implicitly added libraries which will not be removed automatically at a device update.

Converting Devices

Converting Devices

Introduction

EcoStruxure Machine Expert allow you to convert a device that is configured in your project to a different, but compatible device. The currently configured device is converted into the selected device and the changes that are made are displayed in the **Messages** view.

The **Convert Device** command may automatically add or remove modules. These hardware changes also have influences on the addressing and the libraries.

To help to avoid unintended behavior after a device is converted:

- Verify that the new device supports all functions and communication ports that are required in your project
- Avoid using direct addresses in your application
- Perform a backup of the project to the PC before converting a device
- Reload the project.

⚠WARNING

UNINTENDED EQUIPMENT OPERATION

- Verify that any direct addresses used in your application (for example, `%IB5`) have been converted correctly after device conversion.
- Verify that the modified project contains the intended configurations and provides the intended functionality after you have converted the device.

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

NOTICE

LOSS OF DATA

Perform a backup of the project to the PC before converting a device.

Failure to follow these instructions can result in equipment damage.

Converting a Device

To convert a device to a compatible device, proceed as follows:

Step	Action
1	Perform a backup of the project to the PC by executing the **File > Save Project As...** command before converting a device.
2	Right-click the device you want to convert in the **Devices Tree**.
3	Execute the **Convert Device** command from the contextual menu. **Result**: The **Convert Device** dialog box is displayed. It lists those devices that are compatible with the device you selected and provides further information on the selected device.
4	Select the device from the list in which you want to convert your currently configured device. To display the available versions of a device, select the option **Display all versions (for experts only)**.
5	**NOTE:** If you have not yet performed a backup of your project, click **Cancel** to stop without changes and perform a backup before you start the procedure once again. To start the conversion, click **OK**. **Result**: The currently configured device is converted into the device you selected from the list. The information you entered is kept if the related modules are still available. Any modifications or configurations that could not be converted are listed in the **Messages** view.
6	Verify whether the converted project still contains the intended configurations and provides the intended functions. If not, adapt the configuration or restore the backup of the unchanged project file.

Conversion Information in the Messages View

The following information is displayed in the **Messages** view for the conversion process:

- The source devices and the target devices they have been converted to.
- The parameters that have not been transferred to the target.
- The devices that have not been converted.

To save the information displayed in the **Messages** view, you can copy it to the Clipboard (press CTRL + C) and paste it to a data file (press CTRL + V).

Converting Projects

Converting SoMachine Basic and Twido Projects

Introduction

You can convert a SoMachine Basic or TwidoSoft/TwidoSuite project and the configured controller to a selectable EcoStruxure Machine Expert logic controller, page 876. The controller and the corresponding logic are converted and integrated in the EcoStruxure Machine Expert project.

For the conversion process, execute the **File > Convert SoMachine Basic Project** or the **File > Convert Twido Project** command. The **Convert SoMachine Basic Project** dialog box or **Convert Twido Project** dialog box opens. If the commands are not available, you can insert them in a menu of your choice by using the **Tools > Customize** dialog box (see Menu Commands, Online Help).

If you convert a SoMachine Basic project that was created with a SoMachine Basic version that is newer than the latest supported version, this is indicated by a message in the **Messages** view (see Menu Commands, Online Help). You can then continue or cancel the conversion. If you continue, the application will be converted, but it may not be possible to do so without encountering errors that will need to be rectified. In this case, review and verify both the message view and your application before attempting to put it into service.

NOTE: Verify that the SoMachine Basic or Twido project is valid before you convert it into EcoStruxure Machine Expert.

NOTE: It is not possible to convert password-protected projects.

To help to avoid unintended behavior after a project was converted, verify that the target controller supports the functions and communication ports that are required in your project.

⚠WARNING

UNINTENDED EQUIPMENT OPERATION

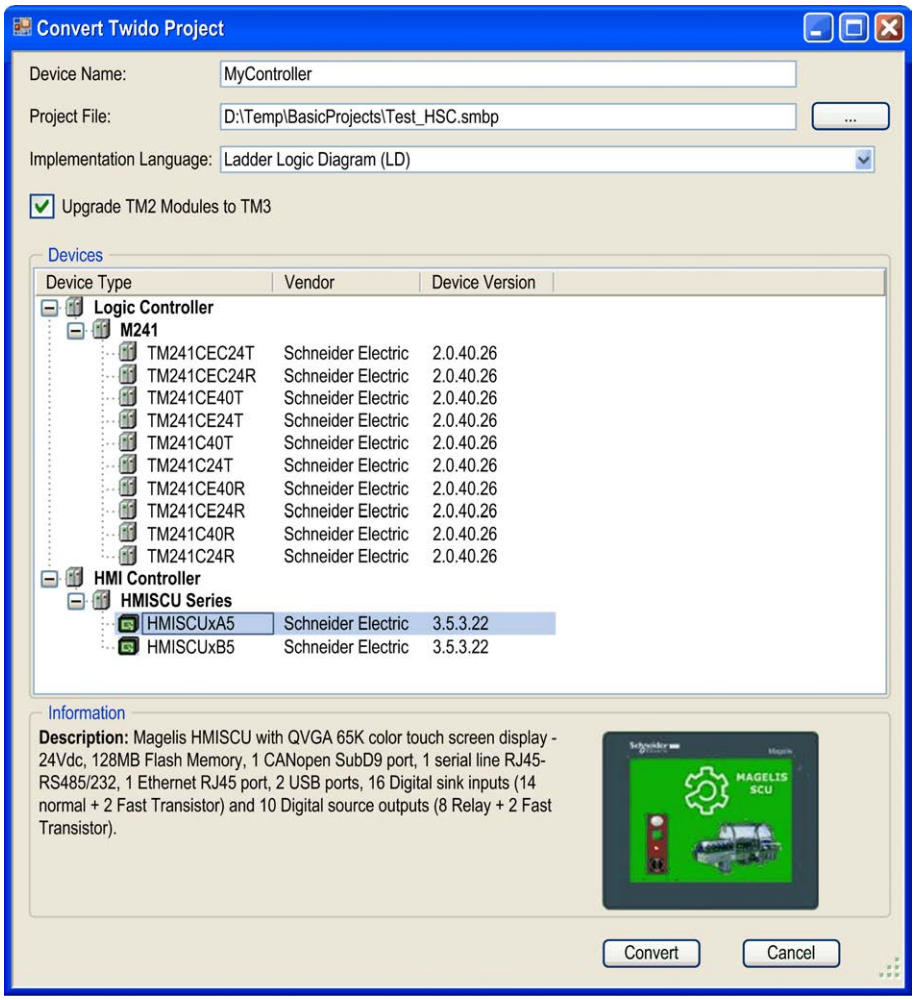
- Verify that the program for the target controller contains the intended configurations and provides the intended functions after you have converted the project.
- Fully debug, verify, and validate the functionality of the converted program before putting it into service.
- Before converting a program, verify that the source program is valid, i.e., is downloadable to the source controller.

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

NOTE: For more information, advice and important safety information concerning importing projects into EcoStruxure Machine Expert, see the *Compatibility and Migration User Guide*.

Converting a SoMachine Basic or a Twido Project

To convert a SoMachine Basic or a Twido project, proceed as follows:

Step	Action
1	To start the conversion process, perform one of the three actions in the EcoStruxure Machine Expert Logic Builder (as listed in the *Introduction* block of this chapter, page 69). **Result**: The **Convert SoMachine Basic Project** dialog box or **Convert Twido Project** dialog box opens: 
2	Enter a name for the controller in the **Device Name** field.
3	Enter the path to the SoMachine Basic or Twido project file in the **Project File** box, or click the **...** button to browse for the file. **NOTE:** If you already browsed for your SoMachine Basic or Twido project in the **Open project** dialog box, the path has been entered automatically in the **Project File** field and cannot be edited.
4	Select the programming language in which the logic will be converted from the **Implementation Language** list. The following programming languages are supported: • Ladder diagram (LD) • Function block diagram (FBD) • Instruction list (IL) • Continuous function chart (CFC)
5	Select the target controller from the **Devices** list in which you want to convert your SoMachine Basic or Twido controller. Further information on the selected device is displayed in the **Information** area of the dialog box.
6	Click **Convert** to start the conversion. **Result**: The SoMachine Basic or Twido project is converted and integrated in the open EcoStruxure Machine Expert project. Modifications or configurations that could not be converted are listed in the **Messages** view (see Menu Commands, Online Help).

Step	Action
7	Consult the category **Project Conversion** of the **Messages** view and verify the errors and alerts detected and listed.
8	Verify whether the converted project contains the intended configurations and provides the intended functions. If not, adapt the configuration.

IEC Compatibility of Object and Variable Names

Object names and variable names in EcoStruxure Machine Expert projects have to comply with the naming conventions defined in the IEC standard. Any names in your SoMachine Basic or Twido project that do not comply with the standard are automatically adapted to IEC conventions by the converter.

If you want to preserve names that are not IEC-compatible in the converted EcoStruxure Machine Expert project, activate the option **Allow unicode characters for identifiers** in the **Project Settings > Compile options** dialog box (see Menu Commands, Online Help).

TwidoEmulationSupport Library

The TwidoEmulationSupport library (see Twido Emulation Support Library, Library Guide) contains functions and function blocks that provide SoMachine Basic and TwidoSoft/TwidoSuite functionality in an EcoStruxure Machine Expert application. The TwidoEmulationSupport library is automatically integrated in the EcoStruxure Machine Expert project with the converted controller.

Conversion of the Application Program

In the target EcoStruxure Machine Expert project, separate programs are created for each SoMachine Basic POU and free POU and for each Twido subroutine and program section. The programming language that is used for these programs is determined by the **Implementation Language** selected in the **Convert SoMachine Basic Project** / **Convert Twido Project** dialog box. An exception is made for POUs that were programmed in graphical Grafcet. They are converted to an SFC program. For detailed information, refer to the Grafcet section in this chapter, page 79.

For each language object (such as memory objects or function blocks) being used by the application program, one global variable is created. Separate global variable lists, page 205 for the different object categories (one for memory bits, one for memory words and so forth) are created.

The following restrictions apply for the conversion of the application program concerning the program structure:

- In EcoStruxure Machine Expert, it is not possible to jump to a label, page 294 in another program.
- It is not possible to define Grafcet steps in a subprogram.
- It is not possible to activate or deactivate Grafcet steps (per `#` and `D#` instruction) in a subprogram.

Conversion of Memory Objects

The areas provided for memory objects in SoMachine Basic and Twido differ from EcoStruxure Machine Expert.

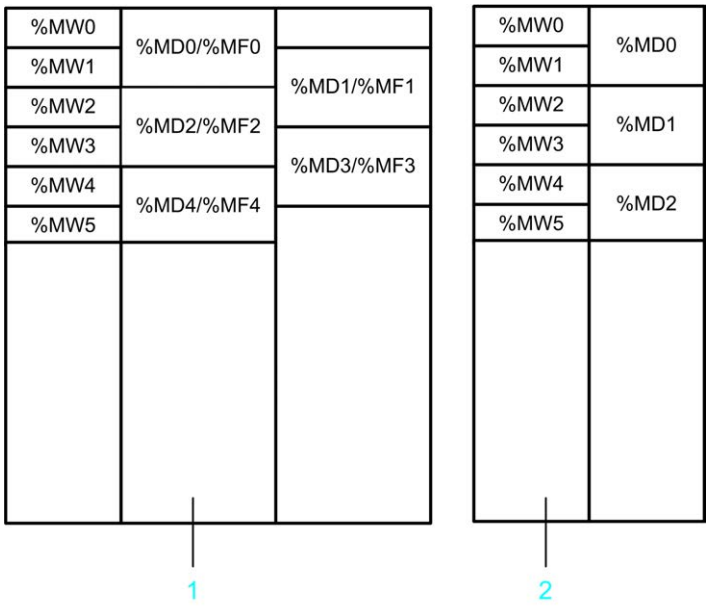
In SoMachine Basic and Twido, there are three distinct areas for memory objects:

Area	Memory objects included
memory bit area	memory bits (%M)
memory word area	• memory words (%MW) • double words (%MD) • floating point values (%MF)
constant area	• constant words (%KW) • double words (%KD) • floating point values (%KF)

In EcoStruxure Machine Expert, there is only the memory word area for memory objects:

Area	Memory objects included
memory word area	• memory words (%MW) • double words (%MD) • floating point values There is no specific addressing format for floating point values. Floating point variables can be mapped on a %MD address.

The graphic provides an overview of the different layouts of %MD and %MF addresses in SoMachine Basic / Twido and EcoStruxure Machine Expert.



1 Memory addresses in SoMachine Basic / Twido

2 Memory addresses in EcoStruxure Machine Expert

Memory objects are converted as follows:

Source memory objects	Target memory object	Further information
%MW	Mapped to the same %MW address **Example** %MW2 is mapped on %MW2.	For each %MW object, a global variable of type INT is created.
%MD and %MF with even addresses	Mapped such that they are located on the same %MW address as before. **Example** %MD4 / %MF4 are mapped on %MD2.	For each %MD object, a global variable of type DINT is created. For each used %MF object, a global variable of type REAL is created.
%MD and %MF with uneven addresses	Cannot be mapped because a DINT variable cannot be located on an odd word address.	A variable is created to help ensure that the converted application can be built. However, you need to examine the effect that such variable creation has on the overall functionality of your program.
%M	Mapped as a packed bit field to a fix location in the %MW area.	For each %M object, a global variable of type BOOL is created.
%KW	Mapped to consecutive addresses of the %MW area.	For each used %KW object, a global variable of type INT is created.

The relationship between %KW, %KD, and %KF objects is the same as for %MW, %MD, and %MF objects. For example, %KD4 / %KF4 are mapped on the same location as %KW4. Uneven %KD / %KF addresses cannot be mapped.

Remote Access

Memory objects (%MW, %MD, %MF, and %M) can be accessed by a remote device through Modbus services:

- If a remote device accesses %MW, %MD or %MF objects in the source application, this access will still be available in the EcoStruxure Machine Expert application.
- If a remote device accesses %M objects in the source application, this access will no longer be available in the EcoStruxure Machine Expert application.

Handling Rising and Falling Edges

A rising/falling edge contact is converted as follows:

1. An additional global variable with the suffix *_Rise/_Fall* is created (for example, *M1_Rise* for a rising edge contact for *%M1*).
2. This variable is set via an *R_TRIG/F_TRIG* instance in the *SystemFunctions* program.

Edge detection is performed at the beginning of the controller cycle.

A *FALLING/RISING* instruction is directly converted into an *R_TRIG/F_TRIG* instance.

Edge detection is performed at the same place of the execution sequence as in the original application.

Conversion of Function Blocks

For the following function blocks in SoMachine Basic / Twido, the TwidoEmulationSupport library provides function blocks with compatible functions:

SoMachine Basic / Twido function block	TwidoEmulationSupport library function block
Timers %TM	FB_Timer
Counters %C	FB_Counter
Register %R	FB_FiFo / FB_LiFo
Drum %DR	FB_Drum
Shift bit register %SBR	FB_ShiftBitRegister
Step counter %SC	FB_StepCounter
Schedule %SCH	FB_ScheduleBlock
PID	FB_PID
Exchange / message %MSG	FB_EXCH
High-speed counter %HSC / %VFC	They are converted as described in the section *Conversion of Fast Counters, High-speed Counters (*Twido*: Very Fast Counters) and Pulse Generators*, page 77 of this chapter.
Fast counter %FC	
PLS pulse generator %PLS	
PWM pulse generator %PWM	
PTO function blocks %PTO, %MC_xxx_PTO	
Frequency generator %FREQGEN	
Communication function blocks READ_VAR, WRITE_VAR, WRITE_READ_VAR , and SEND_RECV_MSG	FB_ReadVar, FB_WriteVar, FB_WriteReadVar, and FB_SendRecvMsg
SMS function block SEND_RECV_SMS	They are not converted.
MC_MotionTask_PTO	
Drive function blocks %MC_xxx_ATV	
%DATALOG	

For the conversion of function blocks, note the following:

- The TwidoEmulationSupport library does not provide any function blocks for hardware-related functions, such as high-speed counters, fast counters, and the pulse generators. They must be controlled through function blocks provided by the platform-specific HSC and PTO_PWM libraries. These function blocks are not compatible with the source function blocks. In short, a full conversion is not possible if the source program contains functions based on controller hardware resources. For further information, refer to the description *Conversion of Fast Counters, High-speed Counters (Twido: Very Fast Counters) and Pulse Generators*, page 77.
- In SoMachine Basic / Twido, the messaging function is provided by the EXCHx instruction and the %MSGx function block. In the EcoStruxure Machine Expert application, this function is performed by a single function block FB_EXCH.
- In SoMachine Basic / Twido, certain function blocks can be configured using special configuration dialog boxes. This configuration data is provided to the function blocks of the TwidoEmulationSupport library by dedicated parameters.
- If a rung contains multiple function blocks, the converter may split the rung into multiple logic networks.

Conversion of Network Objects

The table indicates the network object types that are supported by the conversion:

Network object	Object function	Supported
%QWE	Input assembly (EtherNet/IP)	Yes
%IWE	Output assembly (EtherNet/IP)	Yes
%QWM	Input registers (Modbus TCP)	Yes
%IWM	Output registers (Modbus TCP)	Yes
%IN	Digital inputs (IO scanner)	Only Serial IO scanner
%QN	Digital outputs (IO scanner)	Only Serial IO scanner
%IWN	Input registers (IO scanner)	Only Serial IO scanner
%QWN	Output registers (IO scanner)	Only Serial IO scanner
%IWNS	(IO scanner Diagnostics)	Only Serial IO scanner

Conversion of System Variables

The following system bits and words are converted:

System bit / word	Further information
`%S0`	Is set to 1 in the first cycle after a cold start. **NOTE:** It is not possible to trigger a cold start by writing to this system bit.
`%S1`	Is set to 1 in the first cycle after a warm start. **NOTE:** It is not possible to trigger a warm start by writing to this system bit.
`%S4`	Pulse with the time base 10 ms.
`%S5`	Pulse with the time base 100 ms.
`%S6`	Pulse with the time base 1 s.
`%S7`	Pulse with the time base 1 min.
`%S13`	Is set to 1 in the first cycle after the controller was started.
`%S18`	Is set to 1 if an arithmetic overflow occurs. **NOTE:** This flag is provided by the TwidoEmulationSupport library and is only set by functions provided by this library.
`%S21`, `%S22`	Are only written. Reading is not supported for these variables.
`%S113`	Stops the Modbus Serial IOScanner on serial line 1.
`%S114`	Stops the Modbus Serial IOScanner on serial line 2.
`%SW63...65`	Error code of the `MSG` blocks 1...3.
`%SW114`	Enable flags for the schedule blocks.

Other system variables are not supported by the conversion. If an unsupported system variable is used by the source application program, a message is generated in the category **Project Conversion** of the **Messages** view (see Menu Commands, Online Help).

Conversion of Retain Behavior

The variables and function blocks in SoMachine Basic / Twido are retain variables. This means, they keep their values and states even after an unanticipated shutdown of the controller as well as after a normal power cycle of the controller.

This retain behavior is not conserved during conversion. In EcoStruxure Machine Expert, the converted variables and function blocks are regular, which means that they are initialized during unanticipated shutdown and power cycle of the controller. If you need retain variables in your EcoStruxure Machine Expert application, you have to declare this attribute keyword, page 572 manually.

Conversion of Animation Tables

Management of animation tables differs in the source and target applications:

- SoMachine Basic / Twido allow you to define multiple animation lists identified by name. Each animation list can contain multiple entries for objects to be animated. For each variable, you can select the option **Trace**.
- In EcoStruxure Machine Expert, there are 4 predefined watchlists, page 423 (**Watch 1**...**Watch 4**). Each watchlist can contain multiple variables to be animated. One watchlist can contain variables from different controllers.

  For those variables that have the option **Trace** selected in SoMachine Basic / Twido, EcoStruxure Machine Expert creates a trace object. You can view these variables in the trace editor, page 454.

During the conversion process, the entries of the source animation tables are added at the end of watchlist **Watch 1**.

Conversion of Symbols

Symbols defined in a SoMachine Basic / Twido project are automatically transferred into the EcoStruxure Machine Expert project.

The following restrictions apply to the naming of symbols:

If...	Then ...
a symbol name does not comply with the naming rules of EcoStruxure Machine Expert,	the name of the symbol is modified.
a symbol name is equal to a keyword of EcoStruxure Machine Expert,	the name of the symbol is modified.
no variable is created for a language object,	the name of the symbol is discarded.
a symbol is not used anywhere in the application program,	the name of the symbol may be discarded.

For the complete list of symbol modifications that were required, refer to the **Messages** view.

Conversion of Fast Counters, High-Speed Counters (Twido: Very Fast Counters) and Pulse Generators

The function blocks provided by EcoStruxure Machine Expert differ from the function blocks provided by SoMachine Basic / Twido. Nevertheless, the configuration of fast counters, high-speed counters, and pulse generators is converted as far as possible. The following sections provide an overview of the restrictions that apply.

General Restrictions

The following general restrictions apply:

Restriction	Solution
The inputs and outputs used by the converted high-speed counters and pulse generators may differ from the used inputs and outputs of the source application.	Take this into account in the wiring of the converted controller. The reassignment of inputs and outputs is reported in the **Messages** view (see Menu Commands, Online Help).
The SoMachine Basic controller may support a different number of counters and pulse generators than the selected target controller. The conversion function only converts the counters and pulse generators that are supported by the target controller.	You have to adapt your application manually.

Constraints Regarding the Conversion of %FC, %HSC / %VFC, %PLS, and %PWM

For each %FC, %HSC / %VFC, %PLS, and %PWM function block being used in the SoMachine Basic / Twido application, a single program is created in EcoStruxure Machine Expert. You can improve this basic implementation according to the needs of your application.

The following restrictions apply:

Restriction	Solution
The access to function block parameters is performed differently in SoMachine Basic and EcoStruxure Machine Expert. In SoMachine Basic, the parameters of a function block can be accessed directly by the application program, for example, %HSC.P = 100. In EcoStruxure Machine Expert, a controller-specific function block (for example, EXPERTSetParam) has to be used to access a parameter.	If the source application accesses parameters of the function block, you have to extend the converted application accordingly.
The behavior of counters differs in EcoStruxure Machine Expert from SoMachine Basic / Twido when the preset value is set. In Twido: • The down counter continues counting if zero is reached. • The up counter continues counting if the preset value is reached. In EcoStruxure Machine Expert: • The down counter stops counting if zero is reached. • The up counter starts to count from the beginning if the preset value is reached.	You have to adapt your application manually.
The following parameters of SoMachine Basic function blocks cannot be converted to EcoStruxure Machine Expert: Function block %PLS: • Output parameter *D [Done]* • Parameter *R [Duty Cycle]* Function block %PWM: • Parameter *R [Duty Cycle]* Function block %HSC: • Output parameter *U [Counting Direction]*	You have to adapt your application manually.

Constraints Regarding the Conversion of PTO Function Blocks %PTO and %MC_xxxx

For M241:

The PTO function blocks provided by EcoStruxure Machine Expert for M241 controllers are compatible with the PTO function blocks provided by SoMachine Basic. PTO function blocks are converted without restrictions. The only exception is the MC_MotionTask_PTO function block. The MC_MotionTask_PTO is not converted.

For HMISCU:

The PTO function blocks provided by EcoStruxure Machine Expert for HMISCU controllers are not compatible with the PTO function blocks provided by SoMachine Basic. PTO function blocks are not converted.

Constraints Regarding the Conversion of Frequency Generator Function Block %FREQGEN

The frequency generator function block %FREQGEN is converted without restrictions for both M241 and HMISCU controllers.

Conversion of Loop Elements (*FOR* / *ENDFOR*)

The destination languages for the conversion do not support loops. For that reason, a *FOR* loop is broken up into a functionally equivalent sequence of logical networks using label and jump elements.

Conversion of Conditional Elements (*IF* / *ELSE* / *ENDIF*)

The destination languages for the conversion do not support conditional statements (except *EN* / *ENO*, which are already used for other purposes). For that reason, an *IF* structure is broken up into a functionally equivalent sequence of logical networks using label and jump elements.

Conversion of a Grafcet Program

You can write a Grafcet program in a textual or in a graphical way.

Grafcet type	Description	Supported by
Textual	Various IL and LD programming elements are available for the definition, activation, and deactivation of Grafcet states.	• TwidoSoft/TwidoSuite • SoMachine Basic
Graphical	Allows you to draw the layout of steps, transitions, and branches in a graphical manner.	Only SoMachine Basic V1.4 and later versions.

Conversion of Textual Grafcet

The programming languages of EcoStruxure Machine Expert do not support the programming with Grafcet.

For that reason, a converted Grafcet application contains additional language elements that implement the Grafcet management.

Additional element	Description
folder **Grafcet**	This folder contains the following language elements used for the management of the Grafcet state machine.
data structure `GRAFCET_STATES`	This data structure has one bit element for each allowed Grafcet state. If it is an initial state, the element is initialized to TRUE, otherwise it is FALSE.
global variable list **GrafcetVariables**	This global variable list contains the following variables: • 1 variable `STATES` that contains 1 bit for each Grafcet state. Each bit represents the present value of the corresponding Grafcet state (`%Xi` object). • 1 variable `ACTIVATE_STATES` that contains 1 bit for each Grafcet state. If the bit is TRUE, the Grafcet state is activated in the next cycle. • 1 variable `DEACTIVATE_STATES` that contains 1 bit for each Grafcet state. If the bit is TRUE, the Grafcet state is deactivated in the next cycle.
program **Grafcet**	This program implements the Grafcet state machine. It contains the logic for the activation and deactivation of Grafcet steps. The program contains the following actions: • `Init` initializes the Grafcet steps to their initial states. It is executed when the system bit `%S21` is set by the application program. • `Reset` resets the Grafcet steps to FALSE. It is executed when the system bit `%S22` is set by the application program.

The Grafcet instructions in the application program are converted as follows:

- The beginning of each Grafcet step is marked by a label with the name of the step.

  The first statement within the Grafcet step checks if the step is active. If not, it jumps to the label of the next Grafcet step.
- The access to the `%Xi` is converted to an access to the `STATES.Xi` variable.
- A Grafcet activation instruction `#i` is converted to setting the activation bit of state `i` and the deactivation bit of the present state.
- A Grafcet deactivation instruction `#Di` is converted to setting the deactivation bit of state `i` and the deactivation bit of the present state.

You can extend the converted Grafcet program if you consider the information given in this section.

Conversion of graphical Grafcet

Graphical Grafcet is similar to the programming language SFC provided by EcoStruxure Machine Expert. For this reason, a graphical Grafcet POU is converted to an SFC program, as much as possible.

There are the following differences between graphical Grafcet and SFC:

Graphical Grafcet	SFC	Further information
Can have an arbitrary number of initial steps.	Must have exactly one initial step.	If the graphical Grafcet POU has several initial steps, then the converter creates several initial steps in SFC. This has the effect, that the converted application cannot be built without errors being detected. Adapt the converted program.
Activation of multiple steps of an alternative branch is allowed.	Only one step of an alternative branch can be activated.	Verify that the converted program is working as expected.
The output transitions of a step are evaluated right after the step has been executed.	The transitions of the SFC program are evaluated after the execution of the active steps.	Verify that the converted program is working as expected.
The layout of steps, transitions, and branches is relatively free.	The layout of steps, transitions, and branches is more restricted.	The graphical layout is converted to SFC as far as possible. The incompatibilities encountered during the conversion are reported in the **Messages** view. The step actions and transition sections are fully converted. Complete the created SFC as necessary.

A graphical Grafcet POU can be initialized by setting the system bit `%S21`. If this bit is set in the SoMachine Basic project, the converter activates the implicit variable `SFCInit` and uses it to initialize the SFC program.

Conversion of TM2 Expansion Modules to TM3 Expansion Modules

Twido controllers only use TM2 expansion modules. Even though M221 and M241 Logic Controllers can handle TM2 as well as TM3 modules, it is a good practice to use TM3 modules. To convert the TM2 modules used in your Twido project into TM3 modules for the EcoStruxure Machine Expert project, the option **Upgrade TM2 Modules to TM3** is by default selected.

The TM2 expansion modules are converted into TM3 expansion modules as listed in the table:

Source TM2 expansion module	Target TM3 expansion module	Further information
TM2DDI8DT	TM3DI8	–
TM2DAI8DT	TM3DI8A	–
TM2DDO8UT	TM3DQ8U	–
TM2DDO8TT	TM3DQ8T	–
TM2DRA8RT	TM3DQ8R	–
TM2DDI16DT	TM3DI16	–
TM2DDI16DK	TM3DI16K	–
TM2DRA16RT	TM3DQ16R	–
TM2DDO16UK	TM3DQ16UK	–
TM2DDO16TK	TM3DQ16TK	–
TM2DDI32DK	TM3DI32K	–
TM2DDO32UK	TM3DQ32UK	–
TM2DDO32TK	TM3DQ32TK	–
TM2DMM8DRT	TM3DM8R	–
TM2DMM24DRF	TM3DM24R	–
TM2AMI2HT	TM3AI2H	–
TM2AMI4LT	TM3TI4	It is possible that the behavior of the converted temperature module differs from the original module. Verify the converted module.
TM2AMI8HT	TM3AI8	–
TM2ARI8HT	–	This TM2 module is not converted because there is no corresponding TM3 expansion module. You can replace this module by two TM3TI4 modules.
TM2AMO1HT	TM3AQ2	The target TM3 expansion module has more I/O channels than the source TM2 module.
TM2AVO2HT		–
TM2AMM3HT	TM3TM3	–
TM2ALM3LT		It is possible that the behavior of the converted temperature module differs from the original module. Verify the converted module.
TM2AMI2LT	TM3TI4	The target TM3 expansion module has more I/O channels than the source TM2 module. It is possible that the behavior of the converted temperature module differs from the original module. Verify the converted module.
TM2AMM6HT	TM3AM6	–

Source TM2 expansion module	Target TM3 expansion module	Further information
TM2ARI8LRJ	–	This TM2 module is not converted because there is no corresponding TM3 expansion module. You can replace this module by two TM3TI4 modules.
TM2ARI8LT	–	This TM2 module is not converted because there is no corresponding TM3 expansion module. You can replace this module by two TM3TI4 modules.

NOTE: If you are using TM2 as well as TM3 expansion modules in your EcoStruxure Machine Expert project, note their position in the tree structure: If TM3 nodes are located below TM2 nodes in the tree structure, this is detected as a **Build** error in the **Messages** view.

Conversion of Modbus Serial IOScanner

Due to the differences between controller platforms, and especially for connected controller equipment that depend on the proper functioning of the converted program, you must verify the results of the conversion process. Whether or not errors or alerts are detected during the conversion, it is imperative that you thoroughly test and validate your entire system within your machine or process.

⚠WARNING

UNINTENDED EQUIPMENT OPERATION

- Verify that the program for the target controller contains the intended configurations and provides the intended functions after you have converted the project.
- Fully debug, verify, and validate the functionality of the converted program before putting it into service.
- Before converting a program, verify that the source program is valid, i.e., is downloadable to the source controller.

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

Configuration

The IOScanner configuration is completely converted:

- The devices are converted to the **Generic Modbus Slave** device. The source device type is not preserved.
- The device configuration is completely converted. This includes initialization requests, channel settings, and reset variable.

Function Blocks

The drive function blocks for the control of Altivar drives over the Modbus IOScanner (*MC_xxx_ATV*) are not converted.

Status Handling

Since the IOScanner status handling differs for SoMachine Basic and EcoStruxure Machine Expert, these features can only be partly converted. If your application uses IOScanner status information, verify that this logic still works.

IOScanner Status	Further information
Device Status (`%IWNSx`)	Both SoMachine Basic and EcoStruxure Machine Expert provide status information for a slave device, but the status values are different. The status logic is partly converted.
Channel Status (`%IWNSx.y`)	EcoStruxure Machine Expert does not provide status information for single channels. The channel status is converted to the device status.
System words and bits:	
`%S110/%S111` (IOScanner reset)	They are not converted.
`%S113/%S114` (IOScanner stop)	They are converted.
`%SW210/%SW211` (IOScanner status)	They are not converted.

Conversion of Modbus TCP IO Scanner

The configuration of the Modbus TCP IO Scanner is not converted.

Immediate I/O Access

The instructions `READ_IMM_IN` and `WRITE_IMM_OUT` of SoMachine Basic for immediate access to digital local I/O channels are not converted.

For M241 controllers, you can use the functions `GetImmediateFastInput` and `PhysicalWriteFastOutputs` provided by the PLCSystem library, but consider the following differences:

`READ_IMM_IN` and `WRITE_IMM_OUT` instructions (M221 controllers)	**`GetImmediateFastInput` and `PhysicalWriteFastOutputs` functions (M241 controllers)**
Access to all local inputs and outputs.	Access only to fast inputs and outputs.
`WRITE_IMM_OUT` writes a single bit (similar to the read function). `WRITE_IMM_OUT` returns an error code.	`PhysicalWriteFastOutputs` writes fast outputs at the same time. `PhysicalWriteFastOutputs` only returns the information on which outputs have actually been written.
The error codes of `READ_IMM_IN` and `GetImmediateFastInput` differ.	
`READ_IMM_IN` updates the input object (`%I0.x`).	`GetImmediateFastInput` only returns the read value but does not update the input channel.

NOTE: For HMISCU controllers, no equivalent function exists.

Twido Communication Features

The following communication features of Twido are not converted:

- AS Interface
- CANopen
- remote link

If you use these communication features in your Twido application, you have to adapt the EcoStruxure Machine Expert application manually.

During conversion, one variable is created for each related I/O object in order to allow the EcoStruxure Machine Expert application to be built successfully. These variables are collected in separate global variable lists. This helps you in identifying the variables to be replaced.

Detected Errors and Alerts Indicated in the Messages View

If errors or alerts are detected during the conversion process, a message box is displayed, indicating the number of errors and alerts detected. For further information, consult the category **Project Conversion** of the **Messages** view (see Menu Commands, Online Help). Verify each entry carefully to see whether you have to adapt your application.

⚠WARNING

UNINTENDED EQUIPMENT OPERATION

- Verify that the program for the target controller contains the intended configurations and provides the intended functions after you have converted the project.
- Fully debug, verify, and validate the functionality of the converted program before putting it into service.
- Before converting a program, verify that the source program is valid, i.e., is downloadable to the source controller.

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

- A **warning** message indicates an advisory that the conversion process made some adjustments that, in all likelihood, do not have impact on the functions of your application.
- An **error** message indicates that some parts of the application could not be fully converted. In this case, you have to adapt the application manually in order to preserve the same functionality in the target application.
- If the application program makes use of functionality that cannot be completely converted, the converter creates variables for the unsupported language objects. This allows you to compile your application successfully. However, verify this unsupported functionality after the conversion.

To save the information displayed in the **Messages** view, you can copy it to the Clipboard (press CTRL + C) and paste it to a data file (press CTRL + V).

Common Device Editor Dialogs

What's in This Chapter

Device Configuration 85
I/O Mapping 146

Device Configuration

General Information About Device Editors

Overview

The device editor provides parameters for the configuration of a device, which is managed in the **Devices tree**.

To open the device editor for a specific device, do the following:

- Double-click the node of the device in the **Devices tree** or
- Select device in the **Devices tree** and run the **Edit Object** command from the contextual menu or from the **Project** menu.

The **Tools > Options > Device editor** dialog box allows you to make the generic device configuration views invisible, page 121.

This chapter describes the main device editor dialogs. Bus-specific configuration dialogs are described separately.

Main Device Editor Dialogs

The title of the main dialog consists of the device name, for example **MyPlc**.

Depending on the device type, the device editor can provide the following tabs:

Tab	Description
Communication Settings in Controller selection mode, page 87	Configuration of the connection between programming system and a programmable device (controller). The tab that is displayed depends on the mode selected with the parameter **Communication page** in the **Tools > Options > Device editor** dialog box (see Menu Commands, Online Help).
Communication Settings in Simple mode, page 111	
Communication Settings in Classic mode, page 117	
Configuration, page 121	Display or configuration of the device parameters.
Parameter, page 122	Display or configuration of the device parameters of PacDrive controllers.
Applications, page 122	List of applications running on the controller. Refer to the description in the *Program* chapter, page 172.
Synchronized Files, page 123	List of files that are downloaded to the controller with an application download.
Files, page 124	Configuration of a file transfer between host and controller.
Log, page 125	Display of the controller log file.
PLC Settings, page 128	Configuration of: • Application for I/O handling • I/O behavior in stop status • Bus cycle options
Services	Lets you configure the online services of the controller (RTC, device identification).
Users and Groups, page 132	User management concerning device access during runtime.
Access Rights, page 137	Configuration of the access rights on runtime objects and files for the particular user groups.
Symbol Rights, page 142	Access rights of individual user groups to symbols (symbol sets) on the device.
Task Deployment, page 143	Display of inputs and outputs assigned to the defined task - used for troubleshooting.
Ethernet Services	**Ethernet_1** and **Ethernet_2** tabs summarize the Ethernet connections. The **IP Routing** tab allows you to configure the routes and the cross network transparency through **IP Forwarding** options. The **Ethernet Resources** tab allows you to calculate the number of connections and channels configured.
Status, page 144	Device-specific status and diagnostic messages.
Diagnostic Table	Displays the errors detected by the controller. The data can be accessed using the syntax *NameOfControllerInDeviceTree.NameofParameter*. Example: *MyController.SA_NbPowerOn.*
Information, page 144	General information on the device (for example: name, provider, version).
I/O Mapping, page 146	Mapping of the input and output channels of an I/O device on project (application) variables.
IEC Objects, page 130	Lists objects that allow access to the device from the application. In online mode, it serves as monitoring view.

Tab	Description
OPC UA server configuration	Provides various functionalities to edit the server configuration file *ServerConfig.ini* on the Compact Flash (CF) card of the controller. The OPC UA server configuration allows you to perform the following actions: • Define default properties for new server certificates. • View and delete existing server certificates. • View and add sampling rates. • Import and export server certificates. • Enable and manage various settings. This function is not available for all controllers. Consult the *Programming Guide* specific to your controller.
Emulation Settings, page 144	For configuring the emulation with **EcoStruxure Machine Expert Twin** running in the background and the communication between **EcoStruxure Machine Expert Twin** and the controller.

Communication Settings in Controller Selection Mode

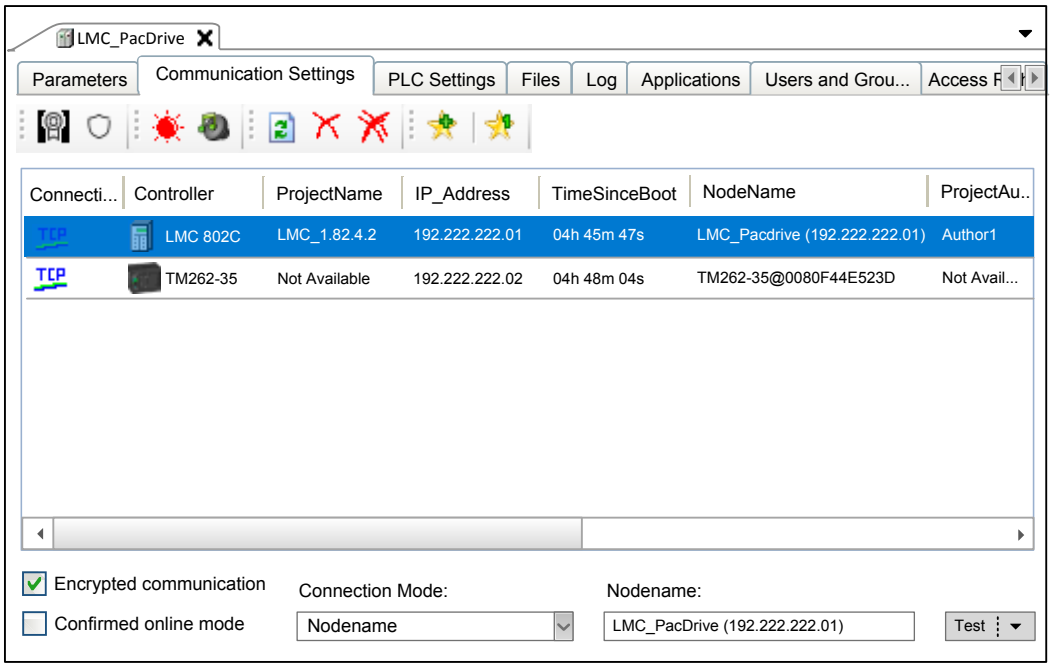
Overview

The **Communication Settings** tab in controller selection mode is displayed when the mode **Controller selection mode** has been selected for the parameter **Communication page** in the **Tools > Options > Device editor** dialog box (see Menu Commands, Online Help). The tab provides access to the Network Device Identification service that allows you to scan the Ethernet network for available controllers and to display them in a list. You can configure the parameters for the communication between the devices (referred to as *controllers* in this chapter) and the programming system.

The list of controllers contains those controllers in the network that have sent a response to the request. It may happen that the controller of your choice is not included in this list. This can have several causes. For causes and suitable solutions, refer to the chapter *Accessing Controllers - Troubleshooting and FAQ*, page 796.

NOTE: Whenever you attempt to gain access to a controller with device user rights management enabled, the credentials are requested.

Communication Settings tab in controller selection mode



The **Communication Settings** tab provides the following elements:

- Buttons in the toolbar
- List providing information on the available controllers
- Options, list, box, and **Test** button at the bottom of the tab

Description of the Buttons in the Toolbar

The following buttons are available in the toolbar:

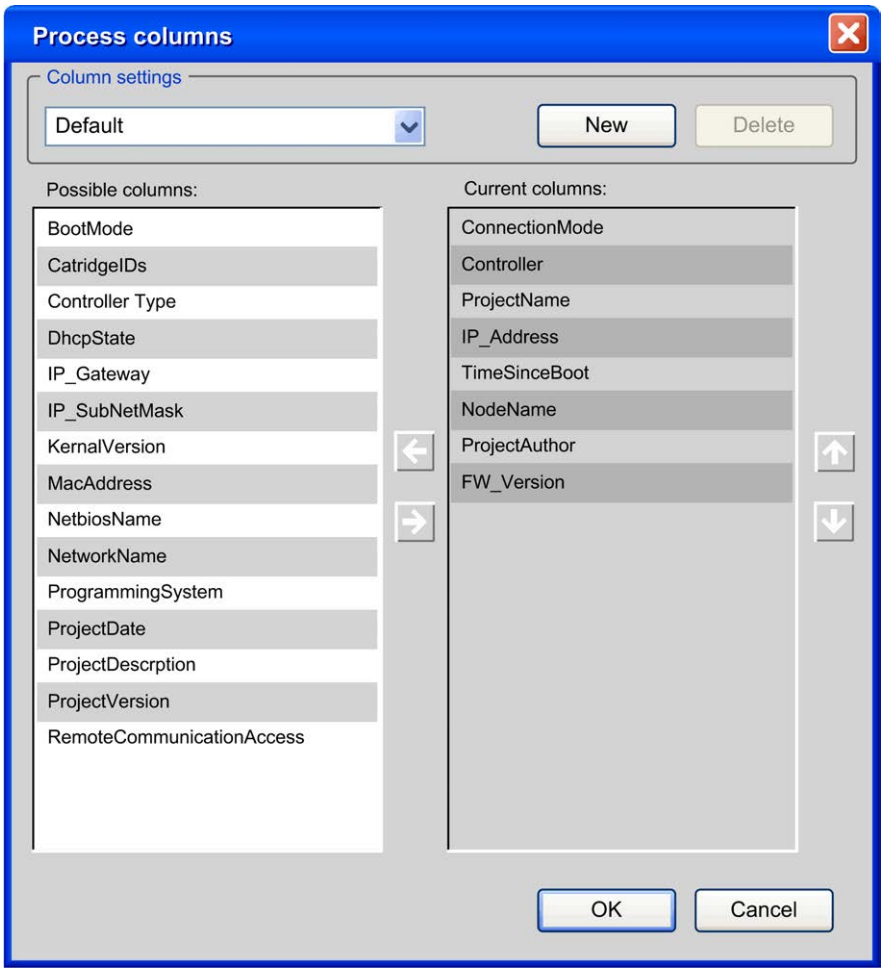
Button	Icon	Description
Change Runtime Security Policy...		Click this button to open the **Change Runtime Security Policy** dialog box, page 115 that allows you to configure the **Communication**, **Code Signing** and **Device User Management** policy.
Security Settings...		Opens the **Device Security Settings** dialog box that displays the security settings of the connected device. It allows you to modify the OpenSSL, user management and app settings in the **Value** column. Click **OK** to write them to the device. For further information about the settings related to password policy, refer to the **Change Runtime Password Policy** dialog box, page 116.
Optical		Click this button to make the selected controller indicate an optical signal: it flashes a control LED. This can help you to identify one controller if several controllers are used. The function stops on a second click or automatically after about 30 seconds. **NOTE:** The optical signal is issued only by controllers that support this function. The command is transmitted to the controller using the **Local Network**. To transmit it with a specific communication mode selected, execute the command from the contextual menu of the controller, page 93 or from the list of the **Test** button, page 99.
Optical and acoustical		Click this button to make the selected controller indicate an optical and an acoustical signal: it starts to beep and flashes a control LED. This can help you to identify one controller if several controllers are used. The function stops on a second click or automatically after about 30 seconds. **NOTE:** The optical and acoustical signals are issued only by controllers that support this function. The command is transmitted to the controller using the **Local Network**. To transmit it with a specific communication mode selected, execute the command from the contextual menu of the controller, page 93 or from the list of the **Test** button, page 99.
Refresh list		Click this button to refresh the list of controllers. A request is sent to the controllers in the network. Controllers that respond to the request are listed with the updated values. Pre-existing entries of controllers are updated with every new request. Controllers that are already in the list but do not respond to a new request are not deleted. They are marked as inactive by a red cross being added to the controller icon. This command is also provided in the contextual menu if you right-click a controller in the list, page 93. To refresh the information of a selected controller, the contextual menu provides the command **Refresh this controller using**. This command requests more detailed information from the selected controller. **NOTE:** The **Refresh this controller using** command can also refresh the information of other controllers.
Remove inactive controllers from list		Controllers that do not respond to a network scan are marked as inactive in the list. This is indicated by a red cross being added to the controller icon. Click this button to remove the controllers that are marked as inactive from the list. **NOTE:** A controller may be marked as inactive. To update the controller state, right-click the controller and execute the command **Refresh this controller using** from the contextual menu.

Button	Icon	Description
Remove all controllers from list		Click this button to remove all controllers from the list and to clear previous connection information from the cache. This command is also available in the contextual menu upon right-clicking a controller. Also refer to Contextual Menu Commands for the Controllers, page 93.
New Favorite... and **Favorite 1**	 	You can use **Favorites** to adjust the selection of controllers to your personal requirements. This can help you keep track of many controllers in the network. A **Favorite** describes a collection of controllers that are recognized by a unique identifier. Click a favorite button (such as **Favorite 1**) to select or deselect it. If you have not selected a favorite, all detected controllers are visible. You can also access **Favorites** from the contextual menu when right-clicking a controller in the list. Move the cursor over a favorite button in the toolbar to view the associated controllers as a tooltip.

List of Controllers

The list of controllers in the middle of the **Communication Settings** tab of the device editor lists those controllers that have sent a response to the network scan. It provides information on each controller in several columns. You can adapt the columns displayed in the list of controllers according to your individual requirements.

To achieve this, right-click the header of a column to open the **Process columns** dialog box.



You can create your own layout of this table. Click **New**, and enter a name for your layout. Shift columns from the list of **Possible columns** to the list of **Current columns** and vice versa by clicking the horizontal arrow buttons. To change the order of the columns in the **Current columns** list, click the arrow up and arrow down buttons.

Connection Modes Indicated in the ConnectionMode Column

The connection mode category is displayed in the column **ConnectionMode**. It is added to the respective controller entry with every connection established to a controller or login. The **Test** button, page 99 allows you to actively add the icon to a controller entry.

Connection mode	Description
ETH	The controller was scanned in the local network or using **Connection Mode** > **Nodename**, page 100 or **Connection Mode** > **IP Address**, page 101.
GAT	The controller was scanned by a given gateway using one of the **via Gateway** connection modes: • **Connection Mode** > **Nodename via Gateway**, page 105 • **Connection Mode** > **IP Address via Gateway**, page 108 The gateway can run either on the local or a different PC.
REM	The controller was scanned by a given gateway using one of the **...via NAT (Remote TCP)** connection modes: • **Connection Mode** > **Nodename via NAT (Remote TCP)**, page 102 • **Connection Mode** > **IP Address via NAT (Remote TCP)**, page 103 The gateway runs on a controller.
TCP	The **Connection Mode** > **IP Address (Fast TCP)**, page 101 was used that connects directly to a controller using TCP.

Contextual Menu Commands for the Controllers

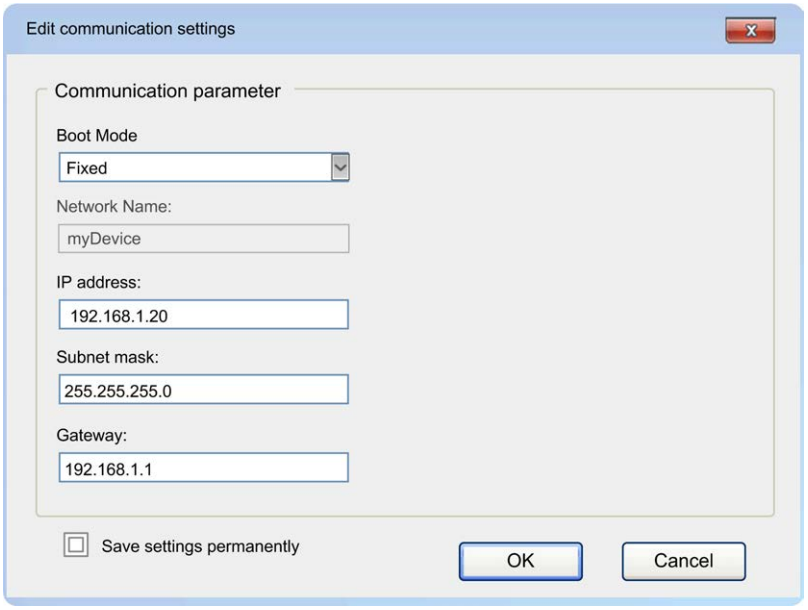
Upon right-clicking a controller row in the list of controllers, the following commands are provided:

Command	Description
Edit communication settings using	Select the command to display a list of the connection modes that are in use for the controller. In addition to the connection modes listed in the Specifying the **Connection Mode**, page 99 section of this chapter, the **Local Network** mode can be available if the controller was detected in the local network. Select an entry to open the **Edit communication settings** dialog box for modifying the communication settings of the selected connection mode. **NOTE:** Controller-specific parameters, such as *RemoteCommunicationAccess* can inhibit access with the **Edit communication settings using** command. Consult the *Programming Guide* specific to your controller for further information.
Remove selected controller from list	The selected controller is removed from the list. Alternatively, select a controller and press the **Delete** key on the keyboard to delete.
Remove all controllers from list	Execute this command to remove all controllers from the list and to clear previous connection information from the cache.
Refresh this controller using	Lists the connection modes that are in use for the controller. Select an entry to send a request to the controller using the selected connection mode, page 99 and to update the displayed values. The **Local Network** mode is available if the controller was detected in the local network. **NOTE:** The **Refresh this controller using Local Network** command can also refresh the information of other controllers.
Refresh list	Execute this command to send a request to the controllers in the network. Controllers that respond to the request are listed with the updated values. Alternatively, press the **F5** key. Pre-existing entries of controllers are updated. Controllers that reside in the local network and are already in the list but do not respond to a new request are not deleted. They are marked as inactive by a red cross being added to the controller icon. In contrast to the **Refresh this controller using** command, this command does not establish a connection to each controller and retrieves less detailed information (not updating, for example, the **TimeSinceReboot** information).
Signal optical using...	Select this command together with the connection mode sub command to send a request to the controller for emitting an optical signal. If the controller is already emitting an optical or acoustical signal when this request is received, it stops both actions. Otherwise it starts flashing rapidly.
Signal optical and acoustical using	Select this command together with the connection mode sub command to send a request to the controller for emitting an optical and acoustical signal. If the controller is already emitting an optical or acoustic signal when this request is received, it stops both actions. Otherwise it starts flashing rapidly and beeping.
Copy to clipboard	Copies the information provided by the selected controller to the clipboard. Alternatively, press the **Ctrl**+**C** keys.
Favorites	Refer to the description of **New Favorite...** and **Favorite 0** in Description of the Buttons in the Toolbar, page 89.

Command	Description
Change device name using	Select this command together with the connection mode sub command to send a request to the controller to display the nodename. If supported by the controller, the nodename can be modified. For further information, refer to Specifying Unique Device Names (**NodeNames**), page 98.
Send echo service using	Select this command together with the connection mode sub command to send an echo service request to the controller. The Logic Builder implements the echo service that is similar to a ping tool. In order to verify the quality of the network connection, five echo data packets are sent to the controller. The amount of user data that is consecutively added to these packets depends on the communication buffer size of the controller. A result message is displayed that indicates the average round-trip delay time and the amount of user data that has been echoed through the connection.

Configuring Communication Settings

To set the parameters for communication between the programming system and a controller, proceed as follows:

Step	Action
1	Select the controller in the list of controllers.
2	Right-click the controller entry, and select the command **Edit communication settings using** with the suitable sub command for the connection mode from the contextual menu. For the available sub commands, refer to Specifying the **Connection Mode**, page 99. When you attempt to gain access to a controller with device user management enabled, a dialog box is displayed requesting the credentials. The additional option **Local Network** is available for controllers that reside in the local network. If supported by the controller, it can be detected with the NetManage UDP broadcast network protocol without credentials and even if the communication settings are invalid. **Result:** The **Edit communication settings** dialog box opens with the settings of the controller.  **NOTE:** Most controllers provide a parameter (such as **RemoteAccess**) that helps prevent changing communication parameters of the controller.

Step	Action
3	Configure the communication parameters: • **Boot Mode** ◦ **FIXED**: A fixed IP address is used according to the values entered below (**IP address**, **Subnet mask**, **Gateway**). ◦ **BOOTP**: The IP address is received dynamically by BOOTP (bootstrap protocol). The values below will be ignored. ◦ **DHCP**: The IP address is received dynamically by DHCP (dynamic host configuration protocol). The values below will be ignored. **NOTE:** Not all devices support BOOTP and/or DHCP. • **IP address** When configuring IP addresses, refer to the hazard message below. This box contains the IP address of the controller. It is a unique address that consists of four numbers in the range of 0...255 separated by periods. The IP address has to be unique in this (sub)network. • **Subnet mask** The subnet mask specifies the network segment to which the controller belongs. It is an address that consists of four numbers in the range of 0...255 separated by periods. Generally, only the values 0 and 255 are used for standard subnet mask numbers. However, other numeric values are possible. The value of the subnet mask is generally the same for all controllers in the network. • **Gateway** The gateway address is the address of a local IP router that is located on the same network as the controller. The IP router passes the data to destinations outside of the local network. It is an address that consists of four numbers in the range of 0...255 separated by periods. The value of the gateway is generally the same for all controllers in the network. • To save the communication settings in the controller even if it is restarted, activate the option **Save settings permanently**.
4	Click **OK** to transfer the settings to the controller.

Carefully manage the IP addresses because each device on the network requires a unique address. Having multiple devices with the same IP address can cause unintended operation of your network and associated equipment.

⚠WARNING

UNINTENDED EQUIPMENT OPERATION

- Verify that all devices have unique addresses.
- Obtain your IP address from your system administrator.
- Confirm that the device's IP address is unique before placing the system into service.
- Do not assign the same IP address to any other equipment on the network.
- Update the IP address after cloning any application that includes Ethernet communications to a unique address.

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

Managing Favorites

To manage favorites in the list of controllers, proceed as follows:

Step	Action
1	Select the controller in the list of controllers.
2	Right-click the controller and select one of the commands: • **New Favorite** to create a new group of favorites. • **Favorite n** in order to ◦ Add the selected controller to this list of favorites ◦ Remove the selected controller from this list of favorites ◦ Remove all controllers from this list of favorites ◦ Select a favorite ◦ Rename a favorite ◦ Remove a favorite

Encrypted communication Option

When the **Encrypted communication** option is selected under the **Communication Settings** tab, communication to the controller will be encrypted.

NOTE: The following prerequisites must be fulfilled to perform encrypted communication with a controller:

- The controller must support TLS (Transport Layer Security).
- A certificate must be available on the controller.

Consult the *Programming Guide* specific to your controller for information on the support of TLS.

The following scenarios are possible when you attempt to log into a controller using encrypted communication:

If...	Then ...	Comment
If the controller does not support TLS (prerequisite 1 is not fulfilled)	Then a message will be displayed when you attempt to log into the controller, indicating that the controller does not support TLS.	The login is denied.
If a certificate is not available on the controller (prerequisite 2 is not fulfilled)	Then a message will be displayed when you attempt to log into the controller, indicating that encrypted communication could not be initialized successfully.	The login is denied.
If both prerequisites are fulfilled	Then a message will be displayed when you attempt to log into the controller for the first time, requesting you to install the (untrusted) controller certificate to the local **Controller Certificates** store of the PC running EcoStruxure Machine Expert.	If you confirm with **OK**: • The login will be successful given that the user password, if required, is provided as well. • Communication to the controller will be encrypted. • The message will not be displayed again. If you click **Cancel**: • The login will be denied. • The message will be displayed with every new attempt to log in.

For further information, refer to the document How To Manage Certificates on the Controller.

Confirmed online mode Option

The **Confirmed online mode** option causes EcoStruxure Machine Expert to display a message requiring confirmation when one of the following online commands is selected: **Force values, Login, Multiple download, Release force**

list, Single cycle, Start, Stop, Write values. To disable the **Confirmed online mode** option and thereby delete the display of this message, clear this option.

Specifying Unique Device Names (NodeNames)

The term **NodeName** is used as a synonym for the term device name. Since nodenames are also used to identify a controller after a network scan, manage them as carefully as IP addresses and verify that each nodename is unique in your network. Having multiple devices assigned the same nodename can cause unpredictable operation of your network and associated equipment.

⚠WARNING

UNINTENDED EQUIPMENT OPERATION

- Ensure that all devices have unique nodenames.
- Confirm that the device's nodename is unique before placing the system into service.
- Do not assign the same nodename to any other equipment on the network.
- Update the nodename after cloning any application that includes Ethernet communications to a unique nodename.
- Create a unique nodename for each device that does not create it automatically, such as M241 and M251 controllers.

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

Depending on the type of controller, the automatic creation of the **NodeName** (device name) may differ in procedure. To create a unique name, some controllers integrate their IP address, others use the MAC address of the Ethernet adapter. In this case, you do not have to change the name.

You can also assign a unique device name (**NodeName**) as follows:

Step	Action
1	Right-click the controller in the list and run the command **Change device name using** from the contextual menu. **Result**: The **Change device name** dialog box opens.
2	In the **Change device name** dialog box, enter a unique device name in the box **New**.
3	Click the **OK** button to confirm. **Result**: The device name you entered is assigned to the controller and is displayed in the column **NodeName** of the list. **NOTE:** Device name and **NodeName** are synonymous.

Specifying the Connection Mode

The **Connection Mode** list at the lower left of the **Communication Settings** tab allows you to select a format for the connection address you have to enter in the **Address** fields.

The following formats are supported:

- Automatic, page 100
- Nodename, page 100
- IP Address, page 101
- IP Address (Fast TCP), page 101
- Nodename via NAT (Remote TCP), page 102 (NAT = network address translation)
- IP Address via NAT (Remote TCP), page 103
- Nodename via Gateway, page 105
- IP Address via Gateway, page 108
- Nodename via MODEM, page 111
- IP Address (PacDriveM only), page 101 (only available in service tools like Controller Assistant)

NOTE: After you have changed the **Connection Mode**, it may be required to perform the login procedure twice to gain access to the selected controller.

NOTE: When you attempt to gain access to a controller with device user management enabled, a dialog box is displayed requesting the credentials.

Testing the Connection for each Connection Mode

For each of the connection modes, a **Test** button is provided. It allows you to establish a test connection to the controller with the selected connection mode. After the test connection has been established, a new row for the controller is added to the list if it is not already displayed. If a row for this controller is already displayed, it is verified whether the icon of the selected connection mode is already indicated in the column **ConnectionMode**. Otherwise, the icon is added. Refer to the list of connection modes, page 92.

Alternatively, you can click the arrow right to the **Test** button to display a list of available commands.

- **Signal optical**
- **Signal optical and acoustical**
- **Send echo service**

The commands perform the same tasks as described in the table Contextual Menu Commands for the Controllers, page 93 when executed upon right-clicking a specific controller.

Connection Mode Automatic

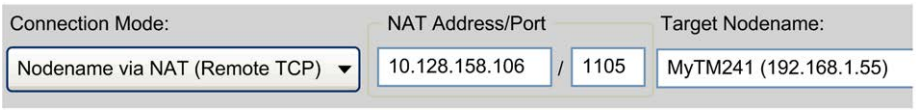
If you select the option **Automatic** from the **Connection Mode** list, you can enter the nodename, the IP address, or the connection URL (uniform resource locator) to specify the **Address**.

NOTE: Do not use spaces at the beginning or end of the **Address**.

If you have selected another **Connection Mode** and you have specified an **Address** for this mode, the address you specified will still be available in the **Address** box if you switch to **Connection Mode > Automatic**.

Example:

Connection Mode > Nodename via NAT (Remote TCP) selected and address and nodename specified



If you switch to **Connection Mode > Automatic**, the information is converted to a URL, starting with the prefix `enodename3://`



If an IP address has been entered for the connection mode, the information is converted to a URL starting with a prefix. For the **Connection Mode > IP Address** , the prefix `etcp3://` is used. For the **Connection Mode > IP Address (Fast TCP)** , the prefix `etcp4://` is used. For example, `etcp4://<IpAddress>`.

NOTE: In the Controller Assistant and the Diagnostics tools, an IP address can additionally have the prefix `etcp2://`. This is only available for PacDrive M controllers.

If a nodename has been entered for the connection mode (for example, when **Connection Mode > Nodename** has been selected), the information is converted to a URL starting with the prefix `enodename3://`. For example, `enodename3://<Nodename>`.

Connection Mode > Nodename

If you select the option **Nodename** from the **Connection Mode** list, you can enter the nodename of a controller to specify the **Address**. The text box is filled automatically if you double-click a controller in the list of controllers.

Example: **Nodename:** `MyM238 (10.128.158.106)`

If the controller you selected does not provide a nodename, the **Connection Mode** automatically changes to **IP Address**, and the IP address from the list is entered in the **Address** box.

Connection Mode > IP Address

If you select the option **IP Address** from the **Connection Mode** list, you can enter the IP address of a controller to specify the **Address**. The box is filled automatically if you double-click a controller in the list of controllers.

Example: **IP Address:** `190.201.100.100`

If the controller you selected does not provide an IP address, the **Connection Mode** automatically changes to **Nodename**, and the nodename from the list is entered in the **Address** box.

> **NOTE:** Enter the IP address according to the format `<Number>.<Number>.<Number>.<Number>`

Connection Mode > IP Address (Fast TCP)

If you select the option **IP Address (Fast TCP)** from the **Connection Mode** list, you can connect to a controller using the TCP protocol. Enter the **Target IP address/Port** of the controller in the respective field. You can adapt the default setting **11740** for the **Port** if you are using network address translation (NAT).

Example: **IP Address:** `190.201.100.100`

> **NOTE:** Enter the IP address according to the format `<Number>.<Number>.<Number>.<Number>`

If the controller is not listed in the list of controllers, click the **Test** button. If the controller sends a response to the network scan, an entry for this controller is added to the list of controllers. This entry is marked by the icon **TCP** being displayed in the first column.

> **NOTE:** This function is not available for all controllers. Consult the *Programming Guide* specific to your controller to find out whether it supports the **IP Address (Fast TCP)** connection mode.

Connection Mode > IP Address (PacDriveM only)

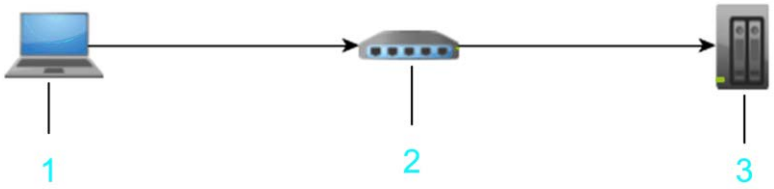
If you select the option **IP Address (PacDriveM only)** from the **Connection Mode** list, you can enter the IP address of a controller to specify the **Address**. The box is filled automatically if you double-click a PacDrive M controller in the list of controllers.

Example: **IP Address:** `190.201.100.100`

> **NOTE:** Enter the IP address according to the format `<Number>.<Number>.<Number>.<Number>`

Connection Mode > Nodename via NAT (Remote TCP)

If you select the option **Nodename via NAT (Remote TCP)** from the **Connection Mode** list, you can specify the address of a controller that resides behind a NAT router in the network. Enter the nodename of the controller, and the IP address or host name and port of the NAT router.



1 PC

2 NAT router

3 Target device

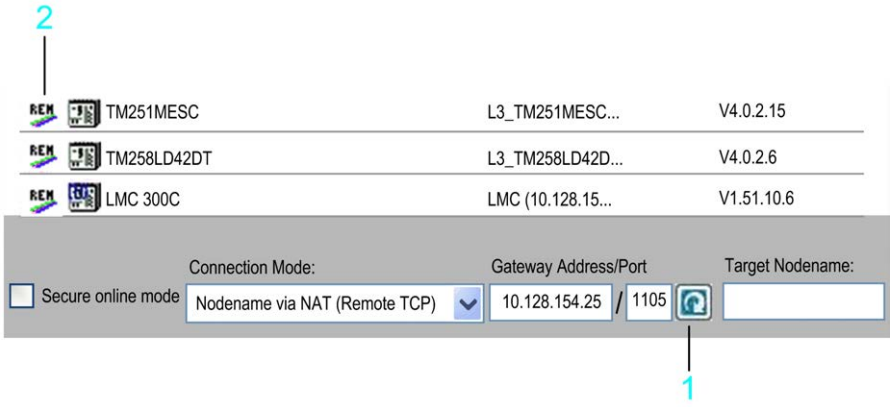
Example: **NAT Address/Port:** `10.128.158.106`**/**`1105` **Target Nodename:** `MyM238 (10.128.158.106)`

> **NOTE:** Enter a valid IP address (format `<Number>.<Number>.<Number>.<Number>`) or a valid host name for the **NAT Address**.
>
> Enter the port of the NAT router to be used. Otherwise, the default port **1105** is used.

The information you enter is interpreted as a URL that creates a remote TCP bridge - using TCP block driver - and then connects by scanning for a controller with the given nodename on the local gateway.

> **NOTE:** The NAT router can be located on the target controller itself. You can use it to create a TCP bridge to a controller.

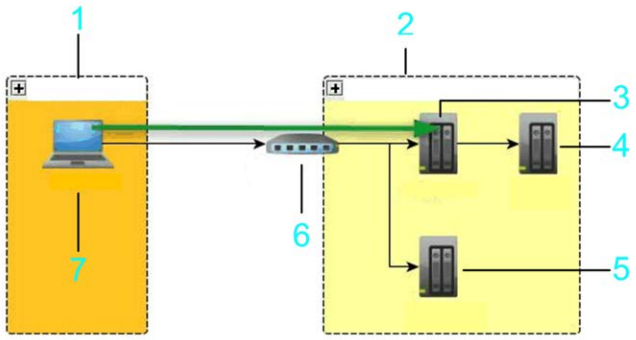
You can also scan a remote network using a remote controller (bridge controller). To achieve this, enter the **NAT Address/Port**, and click the refresh button right to the **NAT Address/Port** text field. The controllers that send a response to the remote network scan are listed in the list of controllers. Each of these entries is marked by the icon **REM** being displayed in the first column. To fill the list with more detailed information, right-click a controller entry and run the command **Refresh this controller**. If the controller supports this function, further information on the controller is added to the list. Consult the *Programming Guide* specific to your controller.



1 Refresh button

2 **REM** icon

In the following example, the **bridge controller**, **controller 2**, and **controller 3** are scanned.



1 Local subnet

2 Remote subnet

3 Bridge controller

4 Controller 3

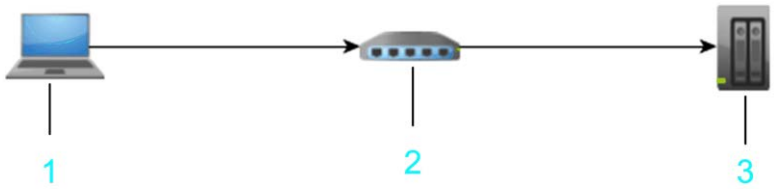
5 Controller 2

6 NAT router

7 PC

Connection Mode > IP Address via NAT (Remote TCP)

If you select the option **IP Address via NAT (Remote TCP)** (NAT = network address translation) from the **Connection Mode** list, you can specify the address of a controller that resides behind a NAT router in the network. Enter the IP address of the controller, and the IP address or host name and port of the NAT router.



1 PC

2 NAT router

3 Target device

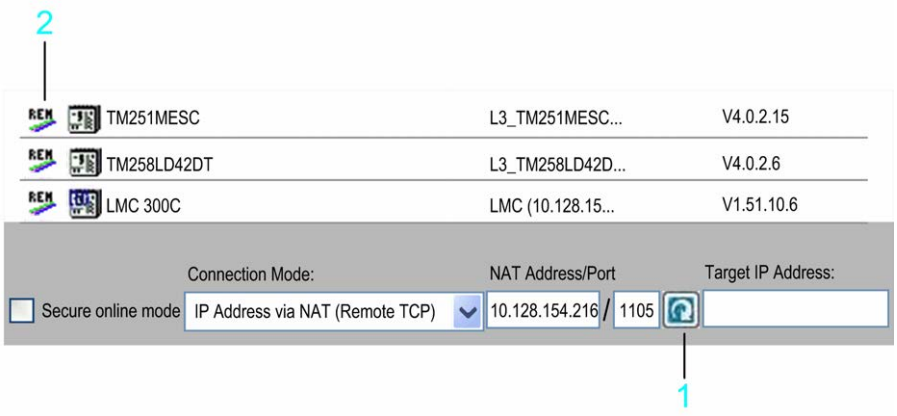
Example: **NAT Address/Port:** `10.128.154.206`**/**`1105` **Target IP Address:** `192.168.1.55`

> **NOTE:** Enter a valid IP address (format `<Number>.<Number>.<Number>.<Number>`) or a valid host name for the **NAT Address**.
>
> Enter the port of the NAT router to be used. Otherwise, the default port **1105** is used.
>
> Enter a valid IP address (format `<Number>.<Number>.<Number>.<Number>`) for the **Target IP Address**.

The information you enter is interpreted as a URL that creates a remote TCP bridge - using TCP block driver - and then connects by scanning for a controller with the given nodename on the local gateway. The IP address is searched in the nodename (such as `MyController (10.128.154.207)`).

> **NOTE:** The NAT router can be located on the target controller itself. You can use it to create a TCP bridge to a controller.

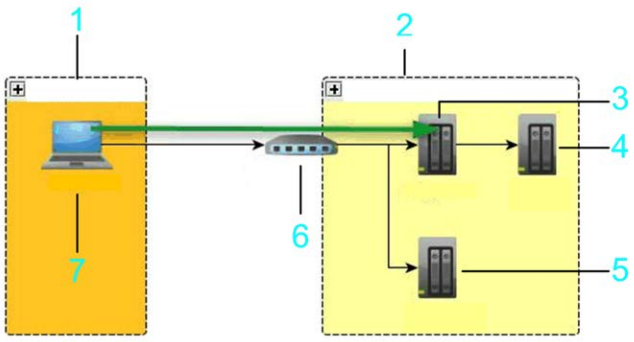
You can also scan a remote network using a remote controller (bridge controller). To achieve this, enter the **NAT Address/Port**, and click the refresh button right to the **NAT Address/Port** text field. The controllers that send a response to the remote network scan are listed in the list of controllers. Each of these entries is marked by the icon **REM** being displayed in the first column. To fill the list with more detailed information, right-click a controller entry and run the command **Refresh this controller**. If the controller supports this function, further information on the controller is added to the list. Consult the *Programming Guide* specific to your controller.



1 Refresh button

2 REM icon

In the following example, the **bridge controller**, **controller 2**, and **controller 3** are scanned.



1 Local subnet

2 Remote subnet

3 Bridge controller

4 Controller 3

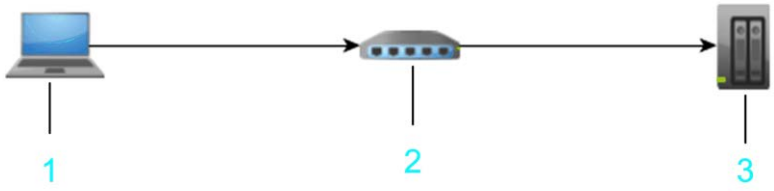
5 Controller 2

6 NAT router

7 PC

Connection Mode > Nodename via Gateway

If you select the option **Nodename via Gateway** from the **Connection Mode** list, you can specify the address of a controller that resides behind or close to an EcoStruxure Machine Expert gateway router in the network. Enter the nodename of the controller, and the IP address or host name and port of the gateway router.



1 PC / HMI

2 PC / HMI / devices with installed EcoStruxure Machine Expert gateway

3 Target device

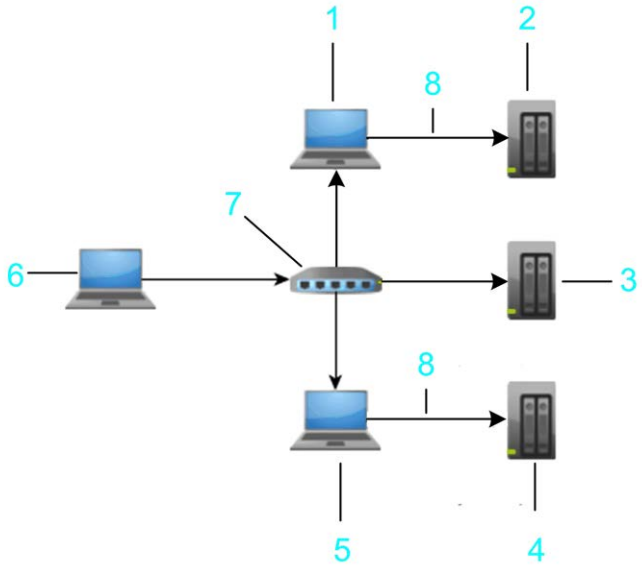
Example: **Gateway Address/Port:** `10.128.156.28`**/**`1217` **Target Nodename:** `MyPLC`

> **NOTE:** Enter a valid IP address (format `<Number>.<Number>.<Number>.<Number>`) or a valid host name for the **Gateway Address/Port:**.
>
> Enter the port of the gateway router to be used. Otherwise, the default gateway port **1217** is used.
>
> Do not use spaces at the beginning or end and do not use commas in the **Target Nodename** box.

The information you enter is interpreted as a URL. The gateway is scanned for a device with the given nodename that is directly connected to this gateway. Directly connected means in the gateway topology it is the root node itself or a child node of the root node.

> **NOTE:** The EcoStruxure Machine Expert gateway can be located on an HMI, destination PC, or the local PC, making it possible to connect to a device that has no unique nodename but resides in a subnet behind an EcoStruxure Machine Expert network.
>
> **NOTE:** To use the **Connection Mode > Nodename via Gateway**, on the PC where the gateway is running, select the option **Enable remote gateway access** in the **Gateway Configuration** tab of the Gateway Management Console. For further information, refer to the Gateway Management Console User Guide.

The graphic shows an example that allows a connection from the PC to the target controller 3 (item 4 in the graphic) by using the address of hop PC2 (item 5 in the graphic) that must have an EcoStruxure Machine Expert gateway installed.



1 Hop PC 1

2 Target controller 1: MyNotUniqueNodename

3 Target controller 2: MyNotUniqueNodename

4 Target controller 3: MyNotUniqueNodename

5 Hop PC 2

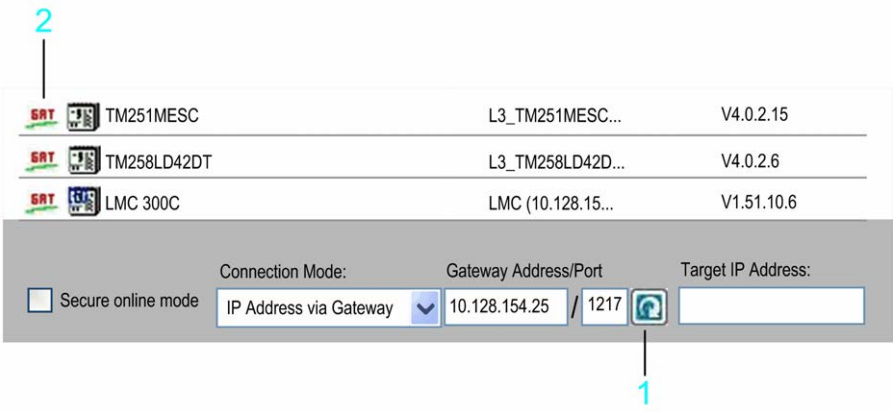
6 PC / HMI

7 Router

8 Ethernet

To verify whether the connection to a specific controller can be established, enter the **Gateway Address/Port**, and click the **Test** button. If the controller sends a response to the network scan, an entry for this controller is added to the list of controllers. This entry is marked by the icon **GAT** being displayed in the first column.

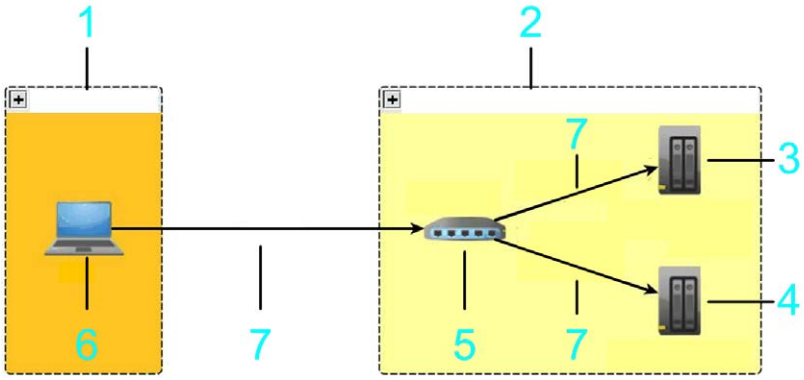
To scan a specific gateway for available controllers, enter the **Gateway Address/Port**, and click the refresh button right to the **Gateway Address/Port** text field. The controllers that send a response to the gateway scan are listed in the list of controllers. Each of these entries is marked by the icon **GAT** being displayed in the first column. To fill the list with more detailed information, right-click a controller entry and run the command **Refresh this controller**. If the controller supports this function, further information on the controller is added to the list. Consult the *Programming Guide* specific to your controller.



1 Refresh button

2 **GAT** icon

The gateway that is scanned can be located on a PC or on an HMI that can reside in the local or in a remote subnet. In the following example, the bridge **target controller 1** and **target controller 2** are scanned.



1 Local subnet

2 Remote subnet

3 Target controller 1

4 Target controller 2

5 Gateway

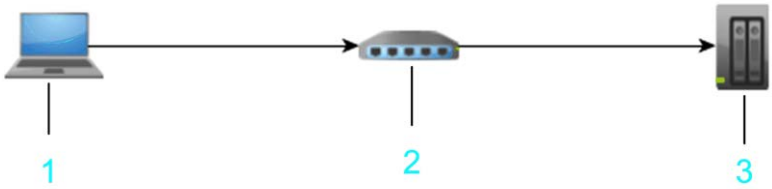
6 PC

7 Ethernet

You can connect to the listed devices using the gateway.

Connection Mode > IP Address via Gateway

If you select the option **IP Address via Gateway** from the **Connection Mode** list, you can specify the address of a controller that resides behind or close to an EcoStruxure Machine Expert gateway router in the network. Enter the IP address of the controller, and the IP address or host name and port of the EcoStruxure Machine Expert gateway router.



1 PC / HMI

2 PC / HMI / devices with installed EcoStruxure Machine Expert gateway

3 Target device

Example: **Gateway Address/Port:** `10.128.156.28`**/**`1217`**Target IP Address:** `10.128.156.222`

NOTE: Enter a valid IP address (format `<Number>.<Number>.<Number>.<Number>`) or a valid host name for the **Gateway Address/Port:**.

Enter the port of the gateway router to be used. Otherwise, the default EcoStruxure Machine Expert gateway port **1217** is used.

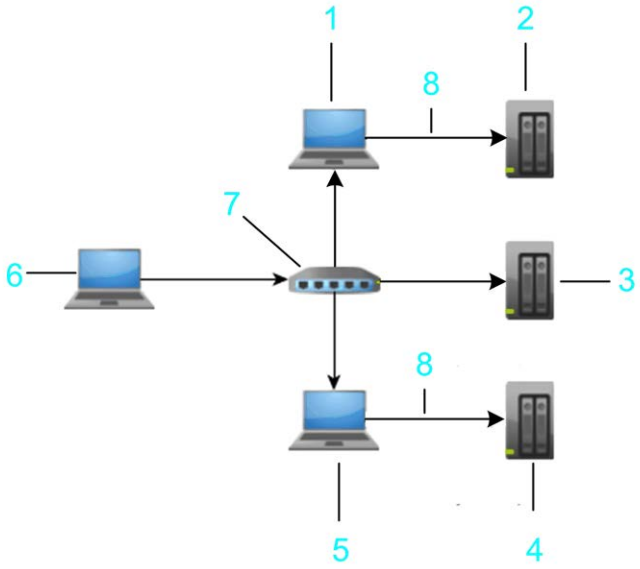
Enter a valid IP address (format `<Number>.<Number>.<Number>.<Number>`) for the **Target IP Address**.

The information you enter is interpreted as a URL. The gateway is scanned for a device with the given IP address. The IP address is searched in the nodename (such as `MyController (10.128.154.207)`).

NOTE: The EcoStruxure Machine Expert gateway can be located on an HMI, destination PC, or the local PC. It is therefore possible to connect to a device that has no unique nodename but resides in a subnet behind an EcoStruxure Machine Expert network.

NOTE: To use the **Connection Mode > Nodename via Gateway**, on the PC where the gateway is running, select the option **Enable remote gateway access** in the **Gateway Configuration** tab of the Gateway Management Console. For further information, refer to the Gateway Management Console User Guide.

The graphic shows an example that allows a connection from hop PC2 (item 5 in the graphic) that must have an EcoStruxure Machine Expert gateway installed to the target controller 3 (item 4 in the graphic).



1 Hop PC 1

2 Target controller 1: 10.128.156.20

3 Target controller 2: 10.128.156.20

4 Target controller 3: 10.128.156.20

5 Hop PC 2

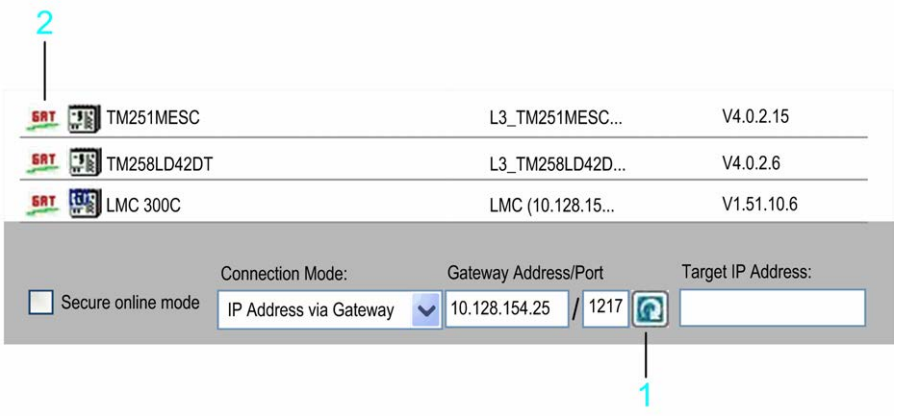
6 PC

7 Router

8 Ethernet

To verify whether the connection to a specific controller can be established, enter the **Gateway Address/Port**, and click the **Test** button. If the controller sends a response to the network scan, an entry for this controller is added to the list of controllers. This entry is marked by the icon **GAT** being displayed in the first column.

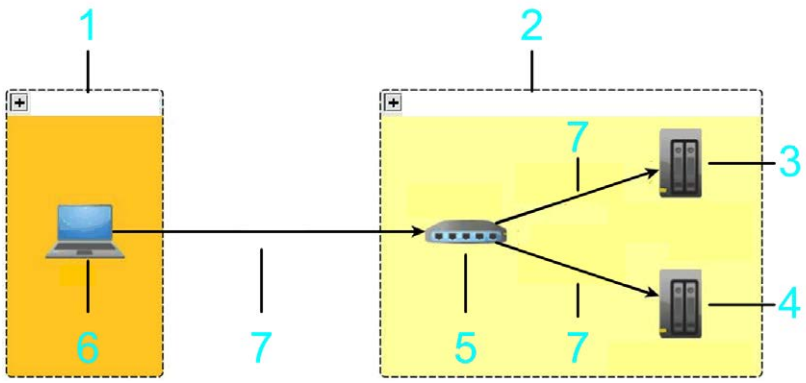
To scan a specific gateway for available controllers, enter the **Gateway Address/Port**, and click the refresh button right to the **Gateway Address/Port** text field. The controllers that send a response to the gateway scan are listed in the list of controllers. Each of these entries is marked by the icon **GAT** being displayed in the first column. To fill the list with more detailed information, right-click a controller entry and run the command **Refresh this controller**. If the controller supports this function, further information on the controller is added to the list. Consult the *Programming Guide* specific to your controller.



1 Refresh button

2 **GAT** icon

The gateway that is scanned can be located on a PC or on an HMI that can reside in the local or in a remote subnet. In the following example, the bridge **target controller 1** and **target controller 2** are scanned.



1 Local subnet

2 Remote subnet

3 Target controller 1

4 Target controller 2

5 Gateway

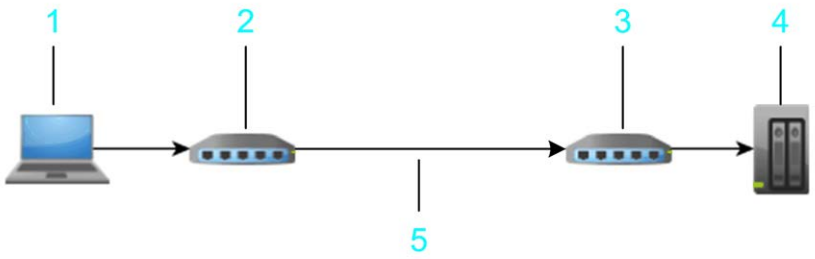
6 PC

7 Ethernet

You can connect to the listed devices using the gateway.

Connection Mode > Nodename via MODEM

If you select the option **Nodename via MODEM** from the **Connection Mode** list, you can specify a controller that resides behind a modem line.



1 PC

2 PC / MODEM

3 Target modem

4 Target device

5 Phone line

To establish a connection to the modem, click the **MODEM > Connect** button. In the **Modem Configuration** dialog box, enter the **Phone number** of the target modem and configure the communication settings. Click **OK** to confirm and to establish a connection to the modem.

If the EcoStruxure Machine Expert gateway is stopped and restarted, any connection of the local gateway is terminated. A message is displayed that has to be confirmed before the restart process is started.

After the connection to the modem has been established successfully, the **MODEM** button changes from **Connect** to **Disconnect**. The list of controllers is cleared and refreshed scanning the modem connection for connected controllers. You can double-click an item from the list of controllers or enter a nodename in the **Target Nodename:** box to connect to a specific controller.

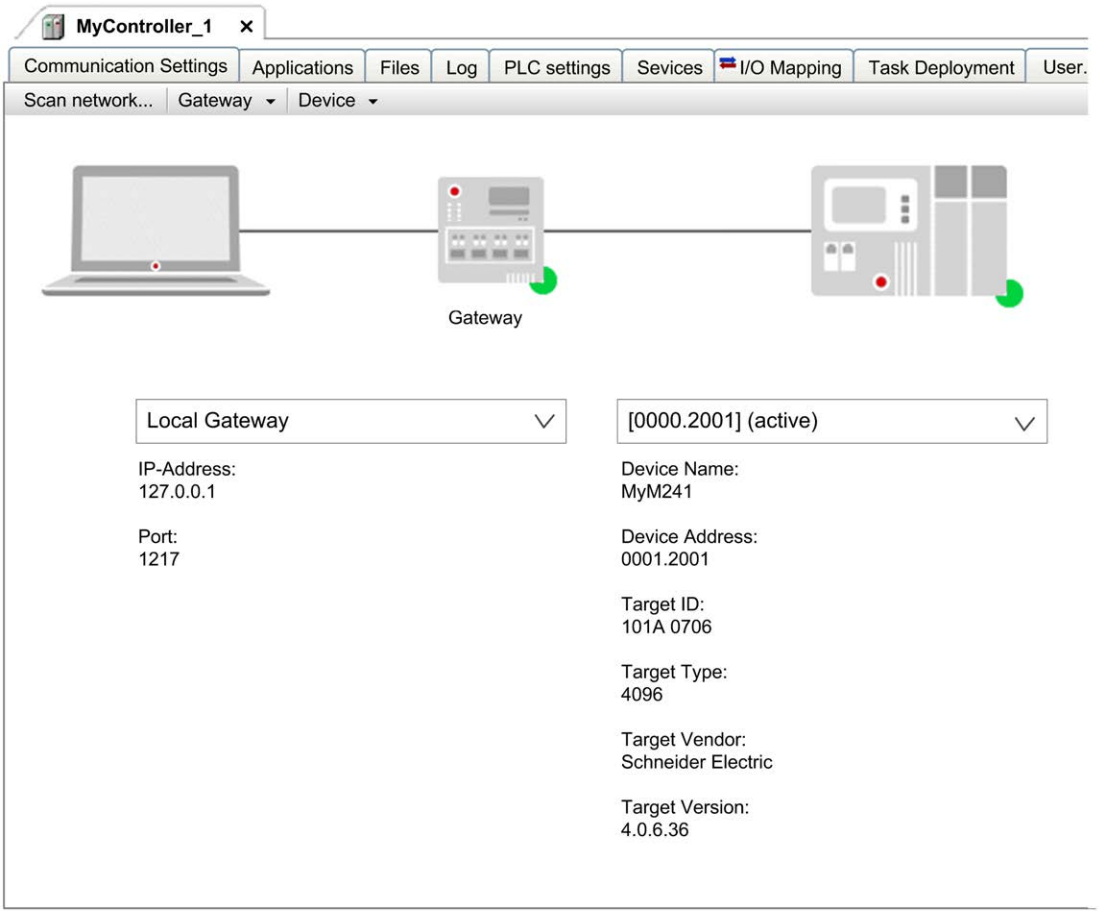
Click the **MODEM > Disconnect** button to terminate the modem connection and to stop and restart the gateway. The list of controllers is cleared and refreshed scanning the Ethernet network.

Communication Settings in Simple Mode

Overview

The **Communication Settings** tab in simple mode is displayed when the mode **Simple mode** has been selected for the parameter **Communication page** in the **Tools > Options > Device editor** dialog box (see Menu Commands, Online Help). It provides a graphic view to configure the parameters for the communication between device and programming system.

The **Communication Settings** tab in simple mode contains an illustration of the programming device, the present gateway, and the target device with the connection status.



Select a gateway and a target device in the selection fields. The list entries you can select are determined by the **Manage gateways** and **Manage favorite devices** parameters.

You can enter the target device in different ways:

- By the IP address, for example **192.168.101.109**
- By the device address, for example **[056D]**
- By the device name, for example **MyDevice**

  NOTE: To search for a device using the device name, unique device names are required within the network.

A status bullet at the right bottom of the gateway symbol indicates the communication status:

Color	Description
Red	Connection cannot be established.
Green	Connection is established.
Black	Connection state is not defined.

NOTE: Some communication protocols do not allow a periodic verification of the gateway. Thus, the status cannot be displayed.

Click the status bullet of the target device to start a network scan for the device. However, this is only possible if the gateway has not already started searching.

Description of the Elements

Elements of the **Communication Settings** tab in simple mode:

Element		Description
Scan network... button		Opens the **Select Device** dialog box that lists the configured gateways and their associated devices. If **Hide non-matching devices, filter by Target ID** is selected, only the devices that have the same target ID as the devices configured in the project are displayed. If **Hide non-matching devices, filter by Target ID** is not selected, all devices detected in the network are displayed.
Gateway list		–
	Add new gateway...	Opens the **Gateway** dialog box for adding a new gateway (see Menu Commands, Online Help).
	Manage gateways...	Opens the **Manage gateways** dialog box showing an overview of all gateways. In this dialog box, you can add or remove gateways. You can change the order of the gateway entries by using the buttons.
	Configure the Local Gateway...	Opens the **Gateway Configuration** dialog box (see Menu Commands, Online Help). It enables you to set up the block driver configuration for the local gateway.
Device list		–
	Add current device to favorites	Adds the defined device to the list of favorite devices.
	Manage favorite devices...	Opens a dialog box showing the list of favorite devices. In this dialog box, you can add or remove devices or change the order of the entries. The device at top of the list defines the default device.
	Rename active device...	Opens a dialog box for renaming the device.
	Wink active device	The connected controller flashes during login.
	Send echo service	The Logic Builder implements the echo service that is similar to a ping tool. In order to verify the quality of the network connection, five echo data packets are sent to the controller. The amount of user data that is consecutively added to these packets depends on the communication buffer size of the controller. A result message is displayed that indicates the average round-trip delay time and the amount of user data that has been echoed through the connection.
	Store communication settings in project	If this option is activated, the communication settings can automatically be restored, even if you are going to open the project on another computer. **NOTE:** If you use the project on another computer, you must reset the active path. If this option is not selected, the settings are stored in the local EcoStruxure Machine Expert options on your computer. In this case, you must reconfigure them if you are going to use the project on another computer. **NOTE:** If you use SVN, make sure to deactivate this option for the device object not to be blocked.

Element		Description
	Confirmed online mode	If this option is activated, you are prompted for confirmation each time one of the following online commands is to be executed: **Force values, Write values, Multiple download, Release force list, Single cycle, Start, Stop**.
	Hide non-matching devices, filter by Target ID	If this option is activated, the list is reduced to those devices which have the same target ID as the device configured in the project.
	Encrypted Communication	If this option is activated, the communication to this controller is encrypted. A certificate is required to log in to the controller. If the certificate is not available, an error message is displayed requesting to display and install the certificate. This option is not available if the option **Security level > Enforce encrypted communication** is selected in the **Security Screen** editor (see Menu Commands, Online Help).
	Change Runtime Security Policy	Opens the **Change Runtime Security Policy** dialog box for modifying the controller settings for the encryption of communication.
	Change Runtime Password Policy	Click this button to open the **Change Runtime Password Policy** dialog box, page 116 that allows you to modify the controller settings regarding the password policy that is being used.
	Security Settings	Opens the **Device Security Settings** dialog box that displays the security settings of the connected device. It allows you to modify the settings in the **Value** column. Click **OK** to write them to the device.

Change Runtime Security Policy Dialog Box

If you select a new communication policy in the **Change Runtime Security Policy** dialog box, the configuration on the controller is modified:

Element	Description
Communication	
Current Policy	Displays the selected policy for the encryption of communication.
New Policy	Contains a list to select the new policy for encryption: • **No encryption**. The controller does not support encrypted communication. • **Optional encryption**: The controller supports encrypted and unencrypted communication. • **Enforced encryption**: The controller supports encrypted communication only.
Code Signing	
Current Policy	Displays the selected policy for code signing.
New Policy	Contains a list to select the new policy for code signing: • **All**: All types of application code are accepted. • **Enforced signing**: Signed application code is accepted (helping to prevent loading an application from untrusted sources).
Device User Management	
Current Policy	Displays the selected policy for user management.
New Policy	Contains a list to select the new policy for user management: • **Optional user management**: The user management on the device is disabled or can be disabled manually. • **Enforced user management**: The user management on the device is enabled and cannot be disabled manually.
Allow anonymous login	If this option is activated, registered components (for example, OPC UA) can connect to the controller without providing credentials. However, the device user management configured for the controller remains active.

Change Runtime Password Policy Dialog Box

If you select a new password policy in the **Change Runtime Security Policy** dialog box, the configuration on the controller is modified:

Element		Description
Password policy is active		If this option is activated, a password policy is active for passwords of user accounts defined in the device user management.
Password settings area		
	Minimum length	Defines the minimum length of the password. Default value: 8
	Number of unique characters	Defines the number of unique characters a password must contain. Default value: 4
	Requires lowercase letter	If this option is activated, the password must contain at least one lowercase letter.
	Requires uppercase letter	If this option is activated, the password must contain at least one uppercase letter.
	Requires digit	If this option is activated, the password must contain at least one digit character.
	Requires special character	If this option is activated, the password must contain at least one special character (such as **#** or **!**).
	Must not contain username	If this option is activated, the password must not contain the username of the logged in user.
Password expiration is active		If this option is activated, users are requested to change their password after a configurable time.
Password expiration settings area		
	Scope	Select the type of users to be considered by the password expiration mechanism: • **ADMINS** • **NONADMINS** • **ALL** (default value)
	Timeout [days]	Select the validity period of the password (in days). Value range: 1...1000 days The default value is controller-specific. To change the password before it expires, the user requires read access for device user management.
Login lock is active		If this option is activated, users are locked after several unsuccessful login attempts according to the parameters configured in the **Login lock settings** area.
Login lock settings area		
	Scope	Select the type of users to be considered by the locking mechanism: • **ADMINS** (default value) • **NONADMINS** • **ALL**

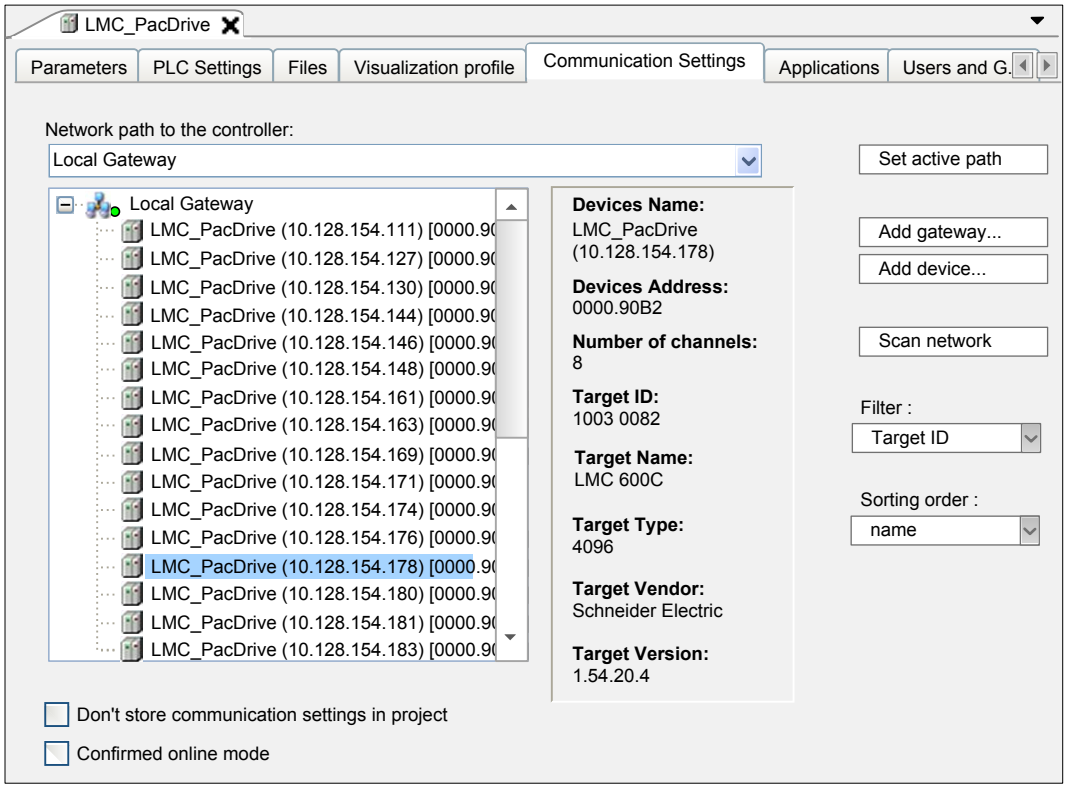
Element		Description
	Maximal Retries	Select the maximum number of unsuccessful login attempts before the user accounts gets locked for the time defined with **Timeout [s]**. If you select the value 0, the number of unsuccessful login attempts is not restricted. The user account is not locked. Default value: 10
	Lock duration [s]	Select the time (in seconds) the user account is locked after the maximum number of unsuccessful login attempts (defined with **Maximal Retries**) have been performed. To unlock a locked user, an administrator or a member of a user group with write permission for the user group of the locked user can assign a new password for the user. Default value: 600

Communication Settings in Classic Mode

Overview

The **Communication Settings** tab in classic mode is displayed when the mode **Classic mode** has been selected for the parameter **Communication page** in the **Tools > Options > Device editor** dialog box (see Menu Commands, Online Help). It allows you to configure the parameters for the communication between device and programming system.

The **Communication Settings** tab in classic mode provides a tree structure to configure the parameters:



This tab is divided in 2 parts:

- The left part shows the configured gateway channels in a tree structure.
- The right part shows the corresponding data and information.

Description of the Tree Structure

When you create the first project on your local system, the local **Gateway** is already available as a node in the tree. This gateway is started automatically during system start.

The settings of this gateway are displayed when you click the **Add gateway...** button:

Example:

Device Name: Gateway-1

Port: 1217

IP-Address: 127.0.0.1

Driver: TCP/IP

A status bullet at the bottom right of the gateway symbol indicates the status of the connection:

Color	Description
Red	Connection cannot be established.
Green	Connection is established.
Black	Connection state is undefined.

NOTE: Some communication protocols do not allow a periodic verification of the gateway. Thus, the status cannot be displayed.

Indented below the **Gateway** node (open/close using the **+/-** sign), you will see entries for the devices which are reachable through this gateway. The device entries are preceded by a  symbol. Entries with a target ID different to that of the device configured in the project, are displayed in gray font. To obtain an up-to-date list of the available devices, use the button **Scan network**.



The device nodes consist of a symbol followed by the node name and the node address. In the right part of the window, the respective **Device Name**, **Device Address**, **Block Driver**, **Encrypted Communication**, **Number of Channels**, **Serial Number**, **Target ID**, **Target Name**, **Target Type**, **Target Vendor**, and **Target Version** are shown.

In the **Network path to the controller** field, the gateway channel is specified automatically by selecting the channel in the tree structure.

NOTE: The parameter **Number of Channels** displays the number of channels that is supported by the selected controller. You cannot monitor online the number of channels that are being used.

A channel is a connection to a client (such as Diagnostics, Logic Builder, WebVisu, OPC, HMI). Depending on the communication service, a client may occupy more than one channel for a short time. When all channels supported by the controller are being used, Logic Builder displays the message **Connection denied by device: All available communication channels are already in use**.

Filter and Sorting Function

You can filter and sort the gateway and device nodes displayed in the tree by the selection boxes in the right part of the tab:

- **Filter**: Allows you to reduce the entries of the tree structure to those devices with a **Target ID** matching that of the device configured in the project.
- **Sorting order**: Allows you to sort the entries of the tree structure according to the **Name** or **Device Address** in alphabetical or ascending order.

Description of the Buttons / Commands

For changing the communication configuration, the following buttons or commands are available:

Button / Command	Description
Set active path	This command sets the selected communication channel as the active path to the controller. See the description of the **Set Active Path** command. Double-clicking the node in the tree structure has the same effect.
Add gateway...	This command opens the **Gateway** dialog box where you can define a gateway to be added to the configuration. See the description of the **Add Gateway** command (see Menu Commands, Online Help).
Add device...	This command opens the **Add Device** dialog box where you can manually define a device to be added to the selected gateway entry (Consider the **Scan network** functionality). See the description for adding devices, page 60.
Edit Gateway...	This command opens the **Gateway** dialog box for editing the settings of the selected gateway. See the description of the **Edit Gateway...** command (see Menu Commands, Online Help).
Delete selected Device	This command removes the selected device from the configuration tree.
Scan for device by address	This command scans the network for devices which have the address specified here in the configuration tree. Those which are found will then be represented in the gateway with the specified node address and their name. The scan refers to devices below that gateway under which an entry is selected. By default, the command is not available in the menus. Add this command with the **Tools > Customize** menu (see Menu Commands, Online Help).
Scan for device by name	This command scans the network for devices which have the names specified here in the configuration tree (case-sensitive search). Those which are found will then be represented in the gateway with the specified name and their unique node address. The scan refers to devices below that gateway under which an entry is selected. By default, the command is not available in the menus. Add this command with the **Tools > Customize** menu (see Menu Commands, Online Help).
Scan for device by IP address	This command scans the network for devices which have the IP address specified here in the configuration tree. Those which are found will then be represented in the gateway with the specified node address and their name. The scan refers to devices below that gateway under which an entry is selected. By default, the command is not available in the menus. Add this command with the **Tools > Customize** menu (see Menu Commands, Online Help).
Send echo service	The Logic Builder implements the echo service that is similar to a ping tool. In order to verify the quality of the network connection, five echo data packets are sent to the controller. The amount of user data that is consecutively added to these packets depends on the communication buffer size of the controller. A result message is displayed that indicates the average round-trip delay time and the amount of user data that has been echoed through the connection.

Button / Command	Description
Configure the local gateway	This command opens a dialog box for the configuration of a local gateway and therefore provides an alternative to manual editing the file *Gateway.cfg*. See the description of the **Configure the local gateway...** command (see Menu Commands, Online Help).
Scan network	This command starts a search for available devices in your local network. The configuration tree of the concerned gateway will be updated accordingly.

Description of the Options

Two options are available below the tree structure:

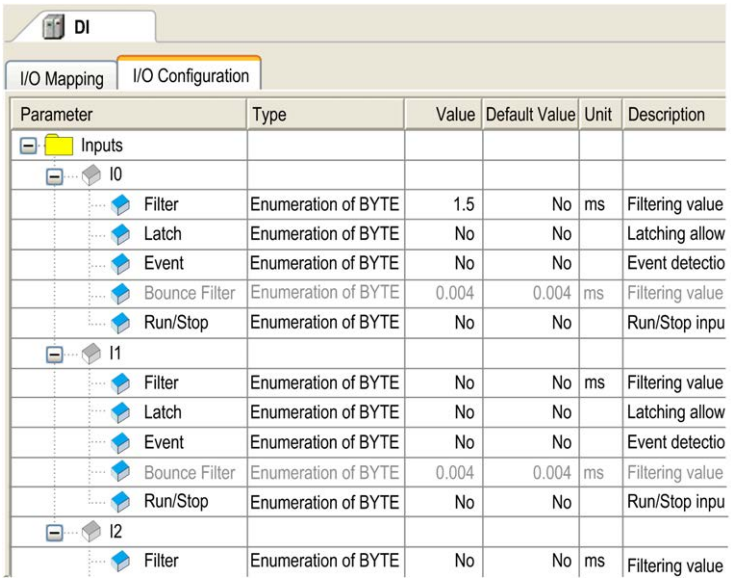
Option	Description
Don't store communication settings in project	Activate this option if the network path definition should not be stored in the project, but in the local option settings on your computer. Therefore, the path setting is restored if the project is reopened on the same computer. It will have to be redefined if the project is used on another system. **NOTE:** If you are using SVN, activate this option to help prevent a lock on the device object.
Confirmed online mode	Activate this option if the user should be prompted for confirmation when selecting one of the following online commands: **Force values, Multiple download, Release force list, Single cycle, Start, Stop, Write values**.

Configuration

Overview

The **Configuration** view is only available in the device editor if the option **Show generic device configuration views** in the **Tools > Options > Device editor** dialog box is activated. The **Configuration** view shows the device-specific parameters, and, if allowed by the device description, provides the possibility to edit the parameter values.

Configuration view of the device editor



The view contains the following elements:

Element	Description
Parameter	Parameter name, not editable
Type	Data type of parameter, not editable
Value	Primarily, the default value of the parameter is displayed directly or by a symbolic name. If the parameter can be modified (this depends on the device description, non-editable parameters are displayed as gray-colored), click the table cell to open an edit frame or a selection list to change the value. If the value is a file specification, the dialog box for opening a file opens by double-clicking the cell. It allows you to select another file.
Default Value	Default parameter value, not editable
Unit	Unit of the parameter value (for example: **ms** for milliseconds), not editable
Description	Short description of the parameter, not editable

Parameters

Overview

The **Parameters** view displays the controller-specific parameters in a table. You can edit the **Value** of those parameters that are defined as editable in the device description. This function is not available for all supported controllers. Consult the *Programming Guide* specific to your controller for further information.

The view contains the following elements:

Element	Description
Parameter	Indicates the parameter name, not editable.
Type	Indicates the data type of the parameter, not editable.
Value	The default value of the parameter is displayed directly or by a symbolic name. If the value can be modified, double-click the field to open an edit frame, a selection list, or a file selection dialog box. Parameters that are not defined as editable in the device description are displayed as gray-colored in this list.
Default Value	Indicates the default parameter value as defined in the device description, not editable
Unit	Indicates the unit of the parameter value (for example: **ms** for milliseconds), not editable.
Description	Indicates the short description of the parameter as defined in the device description, not editable.

Applications

Overview

The **Applications** view of the device editor serves to scan and to remove applications on the controller. Information on the content of the application can be available as well as some details on the application properties.

Description of the Elements

The **Applications** view provides the following elements:

Element	Description
Applications on the PLC	This text box lists the names of applications which have been found on the controller during the last scan (by clicking **Refresh List**). If no scan has been executed yet or if a scan is not possible because no gateway is configured, page 111 for a connection, a message is displayed.
Remove **Remove All**	Click these buttons to remove the application currently selected in the list or all applications from the controller.
Details	Click this button to open a dialog box showing the information as defined in the **Information** tab of the **Properties** dialog box of the application object.
Content	If, in the **View > Properties > Application build options**, the option **Download Application Info** is activated for the application object (see Menu Commands, Online Help), then additional information on the content of the application is loaded to the controller. Click the **Content** button to view the different POUs, in a comparison view. Upon several downloads, this information allows you to compare the code of the new application with that already available on the controller. This provides more information for decisions on how to log in. For further information, refer to the description of the **Login** command.
Refresh List	Click this button to scan the controller for applications. The list will be updated accordingly.

Verifications Before Loading an Application to the Controller

The following verifications are performed before an application is loaded to the controller:

- The list of applications on the controller is compared with those available in the project. If inconsistencies are detected, a dialog box is displayed for either loading the applications not yet available on the controller, or for removing other applications from the controller.
- POUs externally implemented in the application to be loaded are verified as to whether they are also available on the controller. If they are not available on the controller, a message (**unresolved reference(s)**) will be generated in a message box as well as in the **Messages** view if the option **Download** is selected.
- The parameters (variables) of the POUs of the application to be loaded are compared with those of the same-named POUs of the application already available on the controller (validation of signatures). In case any inconsistencies are detected, a message (**signature mismatch(es)**) will be generated in a message box as well as in the **Messages** view if the option **Download** is selected.

  NOTE: If in the **View > Properties > Application build options** the option **Download Application Info** is activated, additional information on the content of the application will be loaded to the controller. See the description of the **Content** button in the previous table.

Synchronized Files

Overview

The **Synchronized Files** view of the device editor lists the files that are downloaded to the controller when the application is downloaded:

- External files, for example, that were added to an application.
- Implicit files, such as the source code archive file, are displayed only in this view if the time of download is configured and the option **Show implicit files for application download on the editor of a PLC** is selected in the **Tools > Options > Device Editor** dialog box (see Menu Commands, Online Help).

This function is not available for all supported controllers. Consult the *Programming Guide* specific to your controller for details on this function.

Description of the Elements

The **Synchronized Files** view provides the following elements:

Element	Description	Example
File name	Name of the file below the application, or direct name of the implicitly transferred file (for example, archive.prj). Double-click the file name to open the file.	archive.prj
Host path	Storage location or original location of the file. Double-click the path to open the directory in the Windows Explorer.	D:\Proj1\Files
Time interval	Time interval for updating the file on the controller.	After download / online change of the application.
Information	Additional information, depending on the object.	Object: external file
Origin	Type of origin of the file.	External file objects, Source code download provider

Files

Overview

The **Files** view of the device editor serves to transfer files between the host and the controller. You can choose any file from a directory of the local network to copy it to the file directory of the currently connected runtime system, or vice versa.

This view is divided in two parts:

- The left part shows the files on the **Host**.
- The right part shows the files on the **Runtime** system.

Description of the Elements

The **Files** view provides the following elements:

Element	Description
	Updates the **Runtime** list.
	Creates a new folder in which you can copy the files.
	Removes the selected files or folders from the list.

Element	Description
Location	Specifies the folder of the respective file system that will be used for the file transfer. Select an entry from the list or browse in the file system tree.
<< >>	File transfer between EcoStruxure Machine Expert and controller (see paragraph below).

For the directories in the file trees there are the following variants:

Element	Description
	Normal file directory.
	Placeholder file directory, defined in the configuration of the runtime system of the controller.
	Implicitly created placeholder file directory of the runtime system of the controller.

File Transfer Between the Controller and EcoStruxure Machine Expert

Step	Action	Comment
1	Select the files to be copied in the file system tree.	Multiple selection is possible. You can even select a folder in order to copy the files contained within.
2	Click the button **>>** or **<<** to transfer the selected files to the directory set in the runtime system.	To transfer in this context means to copy. So, if a file is not yet available in the runtime directory, it will be created there. However, if it already exists, it will be overwritten. If a file with the given name is already available and is write-protected, a corresponding message is generated.

Log

Overview

The **Log** view of the device editor is used to display the events which have been logged on the runtime system of the controller.

This concerns:

- Events at system start or shutdown (loaded components and their versions)
- Application download and boot project download
- Customer-specific entries
- Log entries of I/O drivers
- Log entries of the data server

Description of the Elements

The **Log** view provides the following columns:

Column	Description
Severity	The events of the log are grouped in five categories: • **warning** • **error** • **exception** • **information** • **debug** The buttons in the bar above the listing display the number of loggings in the respective category. Click the buttons to switch on or off the display of the entries of each category.
Time Stamp	Date and time: Example: **13.01.2007 09:48**.
Description	Description of the event: Example: **<Application> started**
Component	Runtime component where the reported event was detected.

The **Log** view provides the following buttons and controls:

Element	Description
	Updates the list.
Components	Choose a particular component in order to obtain only displayed log entries regarding this component. The default setting is **<All components>**.
Logger	Select a logger from the list to display the events it recorded. The selection list provides the available loggings. The default setting is **<Default Logger>**, which is defined by the runtime system. This list is only available if supported by the device. Consult the *Programming Guide* specific to your controller for further information.
.Audit.log	This logger displays the user interactions (such as download, start, stop or set breakpoints) with the controller including the user name. Unsuccessful attempts are logged indicating that users are not allowed to perform a specific action. This function is not available for all supported controllers. Consult the *Programming Guide* specific to your controller for further information.
	Loads the next page with newer log messages.
	Loads the previous page with older log messages.
	Loads the page with the newest log messages and enables automatic scrolling. This is also indicated in the status bar by the message **Auto-Scroll: ON**. When a new log event is detected while the automatic scrolling function is set to OFF, a green asterisk is appended to the button: 
	Loads the page with the oldest log messages.
	Filters logs of type **Warning** and indicates the total number. A blue frame around the button indicates that logs of this type are displayed.
	Filters logs of type **Error** and indicates the total number. A blue frame around the button indicates that logs of this type are displayed.
	Filters logs of type **Exception** and indicates the total number. A blue frame around the button indicates that logs of this type are displayed.
	Filters logs of type **Information** and indicates the total number. A blue frame around the button indicates that logs of this type are displayed.

Element	Description
	Filters logs of type **Debug** and indicates the total number. A blue frame around the button indicates that logs of this type are displayed.
Search in messages	Enter search text. Strings matching the search characters are highlighted in yellow.
	Jumps to the next strings matching the search characters.
	Jumps to the previous strings matching the search characters.
UTC Time	Activate this option to display the time stamp of the runtime system as it is (without conversion). If deactivated, the time stamp of the local time of the computer is displayed (according to the time zone of the operating system). **NOTE:** In order to display the time stamp in UTC (Universal Time Coordinated), you must set the time of the controller to UTC time beforehand (also refer to the **Services** tab of your controller configuration).
	Displays log entries stored in an XML file which may have been exported as described above. The dialog box for browsing for a file opens. The filter is set to **xml-files (*.xml)**. Choose the desired log file. The entries of this file are displayed in a separate window.
	Exports the list to an XML file. The dialog box for saving a file opens. The file filter is set to **xml-files (*.xml)**. The log file is stored with the specified file name with extension .XML in the chosen directory.

Status Bar

The message **Auto-Scrolling:** on the status bar indicates whether the automatic scrolling function is enabled (**ON**) or disabled (**OFF**).

Click the  button to activate the automatic scrolling function.

- **ON**: The **Log** view is refreshed when a new log event is detected.
- **OFF**: When a new log event is detected, the event message is displayed on the status bar and a green asterisk is appended to the button .

Troubleshooting

In case an error is detected and displays the text ***SOURCEPOSITION***, you can open the function by double-clicking or by executing the command **Show source code in editor** from the contextual menu. The cursor marks the relevant position. You can also use the project archive including the **Download information files** and the exported log file for diagnostics. In case of protected functions, the message **The source code is not available for <function name>** is displayed.

PLC Settings

Overview

The **PLC settings** view of the device editor is used to configure general settings for the controller.

Description of the Elements

The **PLC settings** view provides the following elements:

Element	Description
Application for I/O handling	Define here the application assigned to the device in the **Devices tree** that will be monitored for the I/O handling. For EcoStruxure Machine Expert, there is only one application available.
PLC settings area	
Update IO while in stop	Note for PacDrive controllers: Supported by TM5 / TM7 IOs but not by controls or drive based IOs. If this option is activated (default), the values of the input and output channels are updated when the controller is stopped. In case of expiration of the watchdog, the outputs are set to the defined default values.
Behaviour for outputs in Stop	From the selection list, choose one of the following options to define how the values at the output channels are handled in case of controller stop: • **Keep current values** The present values will not be modified. • **Set all outputs to default** The default values resulting from the mapping will be assigned. • **Execute program** You can determine the outputs behavior by a program available within the project. Enter the name of this program here and it will be executed when the controller gets stopped. Click the button **...** to use the **Input Assistant** for this purpose.
Always update variables	Global definition, if the I/O variables are updated in the bus cycle task. This setting takes effect for the I/O variables of the slaves and modules only if their update setting is defined as **Disabled**. • **Disabled (update only if used in a task)**: The I/O variables are only updated if they are used in a task. • **Enabled 1 (use bus cycle task if not used in any task)**: The I/O variables are updated in the bus cycle task if not used in any other task. • **Enabled 2 (always in bus cycle task)**: The variables are updated in every cycle of bus cycle task, regardless of whether they are being used or whether they are mapped to an input or to an output channel. You can set this option separately for each device in the **I/O Mapping** dialog, page 151.
Bus cycle options area	
Bus cycle task	The selection list offers the tasks defined in the **Task Configuration** of the active application (for example, **MAST**). The default setting **MAST** is entered automatically. **<unspecified>** means that the task is selected according to controller-internal settings, which are therefore controller-dependent.

NOTE: Setting the bus cycle task to **<unspecified>** may cause unintended behavior of your application. Consult the *Programming Guide* specific to your controller.

⚠WARNING

UNINTENDED EQUIPMENT OPERATION

Do not set the **Bus cycle task** to **<unspecified>**, unless you understand the bus cycle task settings of the controller.

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

Additional Settings

Element	Description
Generate force variables for IO mapping:	This setting is only available if supported by the device. Consult the *Programming Guide* specific to your controller for further information. If the option is activated, for each I/O channel, which is assigned to a variable in the **I/O Mapping** dialog box, two global variables will be created as soon as the application is built. These variables can be used in an HMI visualization for forcing the I/O value. For further information, refer to the *I/O Mapping* chapter, page 146.
Enable diagnosis for device	The library CAA Device Diagnosis is added to the project automatically. For each device, a function block is generated in the **Applications tree**. If the function block already exists, an extended function block is used (for example, EtherCAT) or an additional function block instance is added. This function block contains the general implementation for the device diagnostics. By use of these function block instances, the status of the devices can be retrieved in the application. Furthermore, detected errors can be evaluated. The library also provides functions for the programmatical evaluation of the **Devices tree** (for example, search in child devices or jump to the parent device). For detailed information, see the PDF document *CAA Device Diagnosis* which is part of the library.
Show I/O warnings as errors	**Warnings** concerning the I/O configuration are detected as errors.
Enable symbolic access for I/Os	This option is only available if supported by the device. Consult the *Programming Guide* specific to your controller for further information. If this option is activated, input and output variables (`VAR_INPUT` and `VAR_OUTPUT`) are automatically created for the input/output channels of the device. For this purpose, an extended function block is created based on the existing function block for each child device. In the application code, direct access to this automatically generated function block is available. This symbolic access works in parallel to the I/O mapping that you configure manually. If this option is not activated, access to I/O channels is not supported. Set up a mapping manually and assign new or existing variables per I/O channel.

IEC Objects

Overview

The **IEC Objects** view of the device editor lists objects that allow access to the device from the application. In online mode, it serves as monitoring view.

If a function block instance is created implicitly for a device, at least this instance is listed as an object in the table. This instance can be used, for example, to restart a bus or to poll information from the application. It depends on the device whether this kind of device instance is available and which access options are provided. Consult the *Programming Guide* specific to your controller for details.

Once the implicit function blocks are created, they must be unique within the application. Creating additional instances of these function blocks may have undesirable results, including malfunctions of the devices they control or disruptions on the industrial network where those devices reside.

⚠WARNING

UNINTENDED EQUIPMENT OPERATION

Do not create another instance of an implicitly created device function block.

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

The table also contains instances of function blocks that are linked to inputs or outputs of the device. The mapping of a function block to a channel is defined in the **I/O Mapping** view, page 146. This view contains the **Go to Instance** button  that provides a direct link to the corresponding object in this **IEC Objects** table.

You can add objects to this table that are not yet linked with a device channel.

In online mode, you can use the **IEC Objects** table as a monitoring view. It indicates the value, the address and the comment for the function block variable at the channel. Additionally, it allows you to write and force values.

Description of the Elements

The **IEC Objects** view provides the following elements:

Element	Description
Add  **Edit** 	Opens the **Select Function Block** dialog box for creating a new instance or for editing the instance selected in the table. Do not create another instance of an implicitly created device function block (refer to the hazard message in the Overview section of this chapter).
Delete 	Deletes the selected entry.
Go to Instance 	Provides a direct link from the selected entry to the corresponding mapping in the **I/O Mapping** view, page 146.
Variable	The object name consists of the device name and the function block name, for example `EL2004_Relay`. Changes to the device name in the project are indicated here. The part of the name after the device name is editable.
Mapping	Mapping type, as in the **I/O Mapping** view, page 146.
Type	Data type, which is the name of the function block in this case.
Value **Prepared value** **Address** **Comment**	Only available in online mode: Indicates the present value, the address, and the comment for the variable at the channel. You can enter a value for writing or forcing the variable.

Users and Groups

Overview

The **Users and Groups** view of the device editor is provided for devices supporting device **Users and Groups** management. If supported by the device, you can view and edit the **Users and Groups** management for the device. Thereafter, you can assign rights to allow certain user groups to access objects on the controller at runtime by assigning **Access Rights**.

For the user management at the project level, refer to the **Project > User Management > Permissions...** command (see Menu Commands, Online Help).

If you want to encrypt your entire project, activate the option **Enable project file encryption** in the **Project Settings > Security** dialog box (see Menu Commands, Online Help).

If you want to encrypt only a part of your code inside the project, put this code inside a compiled library (see Menu Commands, Online Help).

The device **Users and Groups** management can be pre-defined in the device description.

As in the project user management, users have to be members of at least one user group. Only user groups can be assigned specific access rights, page 137.

For managing **Users and Groups**, you have to login as a user with administrative rights.

NOTE: It is not intended that the **Users and Groups** feature be used to protect the EcoStruxure Machine Expert controllers against malicious access, but rather to help prevent mistakes from trusted users.

⚠CAUTION

UNAUTHENTICATED, UNAUTHORIZED ACCESS

- Do not expose controllers and controller networks to public networks and the Internet as much as possible.
- Use additional security layers like VPN for remote access and install firewall mechanisms.
- Restrict access to authorized people.
- Change default passwords at start-up and modify them frequently.
- Validate the effectiveness of these measures regularly and frequently.

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

NOTE: You can use the security-related commands (see Menu Commands, Online Help) which provide a way to add, edit, and remove a user in the online **Users and Groups** management of the target device where you are logged in.

NOTE: You must establish user access-rights using EcoStruxure Machine Expert software. If you have *cloned* an application from one controller to another, you will need to enable and establish user access-rights in the targeted controller.

NOTE: The only way to gain access to a controller that has user access-rights enabled and for which you do not have the password(s) is by performing an **Update firmware** operation using an SD card or USB memory key (refer to the *Controller Assistant User Guide* for further information), depending on the support of your particular controller, or by running a script. Since the process of running a script is specific to each controller, refer to the chapters *File Transfer with SD Card* or *File Transfer with USB Memory Key* in the Programming Guide of the controller you are using. This will effectively remove the existing application from the controller memory, but will restore the ability to access the controller.

Toolbar of the Users and Groups View

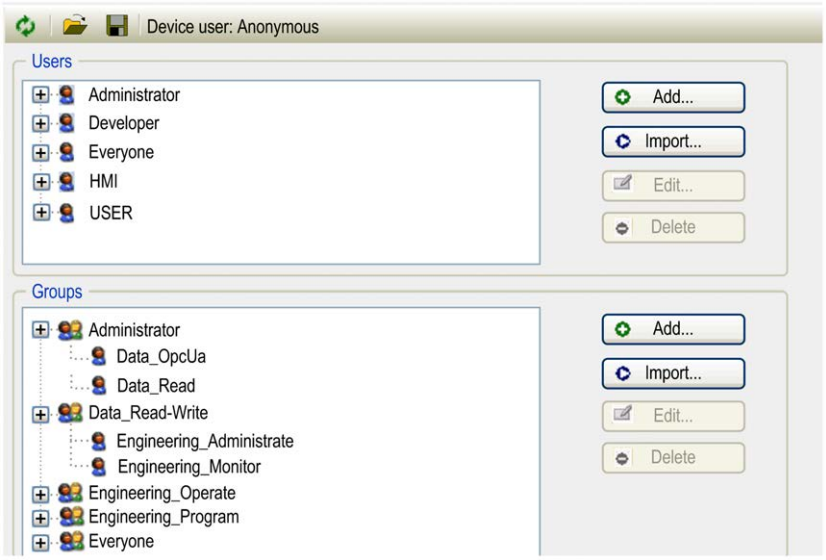
The toolbar provides the following elements:

Element	Description
Synchronization	Click the **Synchronization** button to switch on / off the synchronization between the editor and the **Users and Groups** management in the controller. If **Synchronization** is not activated, then the editor contains a **Users and Groups** management configuration that has been imported from disk, or it does not contain any configuration at all. If **Synchronization** is activated, the data displayed in the editor is continuously synchronized with the **Users and Groups** management configuration on the connected controller. If you invoke **Synchronization** while the editor contains a **Users and Groups** configuration that is not synchronized with the device, you are prompted to decide what will be displayed in the editor: • **Upload from the device and overwrite the editor content**: The **Users and Groups** configuration from the controller is loaded to the editor. The contents of the editor is overwritten. • **Download the editor content to the device and overwrite the user management there**: The configuration from the editor is loaded to the controller. The contents of the controller is overwritten.
Import from disk	**NOTE:** EcoStruxure Machine Expert V2.0 and later versions no longer support *Device user management* files of type *.dum. The import of a file of type *.dum2 overwrites the user management on the device. Afterwards you will be requested to log in to the device as a user using the authentication data of the new user management. When you click the **Import from disk** button, a dialog box opens requesting you to select a file of type *.dum2 from your hard disk. After you have selected the file, the **Enter Password** dialog box opens requesting you to enter the password that was assigned when the file was exported. The user management is enabled. The **Import from disk** is available when you are in offline mode or **Synchronization** is deactivated.
Export to disk	When you click the **Export to disk** button, an **Enter Password** dialog box opens requesting you to enter a password for the device user management file. **NOTE:** The password assigned here will be requested during the **Import from disk** procedure. After you have assigned a password, the dialog box for saving a file is displayed. With EcoStruxure Machine Expert V2.0 and later versions, only files of type *.dum2 are supported.
Device user	Name of the user who is logged into the controller.

Users and Groups Management

The handling of the **Users and Groups** is similar to that of the project user management.

Example of a **Users and Groups** view of the device editor:



This view is divided in 2 parts:

- The upper part is dedicated to access management of **Users**.
- The lower part is dedicated to access management of **Groups**.

Elements of the **Users** section:

Element	Description
The tree structure on the left-hand side lists the defined users and indicates the user groups to which they are assigned as sub nodes.	
Add button	Opens the **Add User** dialog box for creating a new user account. For further information, refer to the section Setting up a New User in the **Users and Groups** Management of the Controller, page 136.
Import button	Opens the **Import Users** dialog box. It lists the user accounts defined in the project user management. Select the entries of your choice (by pressing the **Shift** key for selecting more than one) and click **OK** to import them in the device user management. **NOTE:** For each user to be imported the **Enter Password** dialog box is displayed requesting you to enter the password of the project user account. This password will then also be used for the corresponding device user account.
Edit button	Opens the **Edit User <user name>** dialog box. It allows you to modify the settings of the user account.
Delete button	Deletes the account of the selected user.

Elements of the **Groups** section:

Element	Description
The tree structure on the left-hand side lists the defined groups and indicates the assigned users as sub nodes.	
Add button	Opens the **Add Group** dialog box for creating a new user group. 1. Enter a name for the group. 2. From the list of users, select the users to assign to the new group. 3. Click the **OK** button to confirm and the new group will be displayed in the tree structure.
Import button	Opens the **Import Groups** dialog box. It displays the user groups defined in the project user management. As a prerequisite for importing groups, you must have imported all users that are members of this group into the device user management beforehand. Select the group entries of your choice and click **OK** to import them in the device user management. **NOTE:** The default group **Everyone** cannot be imported in the device user management. **NOTE:** After you have successfully imported a group, assign access rights either: • By adding the new group to an existing group, or • By explicitly assigning access rights to the new group. For further information, refer to Access Rights, page 137.
Edit button	Opens the **Edit Group <group name>** dialog box. It allows you to modify the definition of the group.
Delete button	Deletes the selected group.

Editing or Viewing the Users and Groups Management Before any Users and Groups Have Been Established

If the controller supports device **Users and Groups** management, proceed as follows during first login:

Step	Action	Comment
1	Double-click the controller node in the **Devices tree**.	**Result**: The device editor opens.
2	Select the **Users and Groups** view.	–
3	Click the **Synchronization** button .	**Result**: A dialog box opens prompting you to decide whether the device **Users and Groups** management should be activated.
4	Click **Yes** to confirm the dialog box and to activate device **Users and Groups** management.	**Result**: The **Device user login** dialog box opens.
5	Enter a new user name.	–
6	Enter your individual **Password**.	**NOTE:** The password must comply with the password policy configured in the **Change Runtime Password Policy** dialog box, page 116.
7	Re-enter your individual **Password**.	–
6	Click **OK** to confirm.	**Result**: You are requested to enter the new credentials for accessing the controller. They are assigned the highest user rights level and allow you to manage access rights for users or user groups.

Setting up a New User in the Users and Groups Management of the Controller

If the controller supports device **Users and Groups** management, you can add a new user as follows:

Step	Action	Comment
1	Double-click the controller node in the **Devices tree**.	**Result**: The device editor opens.
2	Select the **Users and Groups** view.	–
3	Click the **Synchronization** button  to load the **Users and Groups** management configuration from the controller to the editor.	If you are not logged in to the controller yet, then the dialog box **Device User Login** opens. It allows you to enter the user name and the password. **Result**: The **Users and Groups** management configuration of the controller is displayed in the editor.
4	Click the **Add** button in the **Users** part of the **Users and Groups** view.	**Result**: The **Add User** dialog box opens.
5	Enter a **Name** for the new user and select a **Default group** for the user from the list.	You can assign the user to other groups later.
6	Enter a new password, confirm the password, and specify whether the user can change the password and whether the user has to change the password at the first login.	**NOTE:** The password must comply with the password policy configured in the **Change Runtime Password Policy** dialog box, page 116.
7	Click **OK** to confirm and to close the **Add User** dialog box.	**Result**: The new user is displayed in the **Users** part as a new node and in the **Groups** part as a new subnode of the selected default group.

Loading a Users and Groups Management From a *.dum File, Modifying it, and Later Downloading it to the Controller

Step	Action	Comment
1	Double-click the controller node in the **Devices tree**.	**Result**: The device editor opens.
2	Select the **Users and Groups** view.	–
3	Click the **Edit** button, browse to the *.dum file that contains the saved **Users and Groups** management, and click **Open** to confirm.	**Result**: The users and groups settings that are saved in the file are displayed in the editor.
4	Adapt the settings according to your requirements.	–
5	Click the **Synchronization** button  to transfer the **Users and Groups** management configuration to the controller.	A dialog box is displayed, prompting you to select an operation.
6	Select the option **Download the editor content to the device and overwrite the user management there**.	**Result**: The **Device user login** dialog box is displayed.
7	Enter login data in order to log in to the controller.	After successful login, the modifications are transferred to the controller. As long as the **Synchronization** button  is activated, modifications made in the editor are automatically transferred to the controller.

Printing the Users and Groups Management Configuration

To print the settings of the **Users and Groups** view, run the command **Print** from the **File** menu or the command **Document** from the **Project** menu.

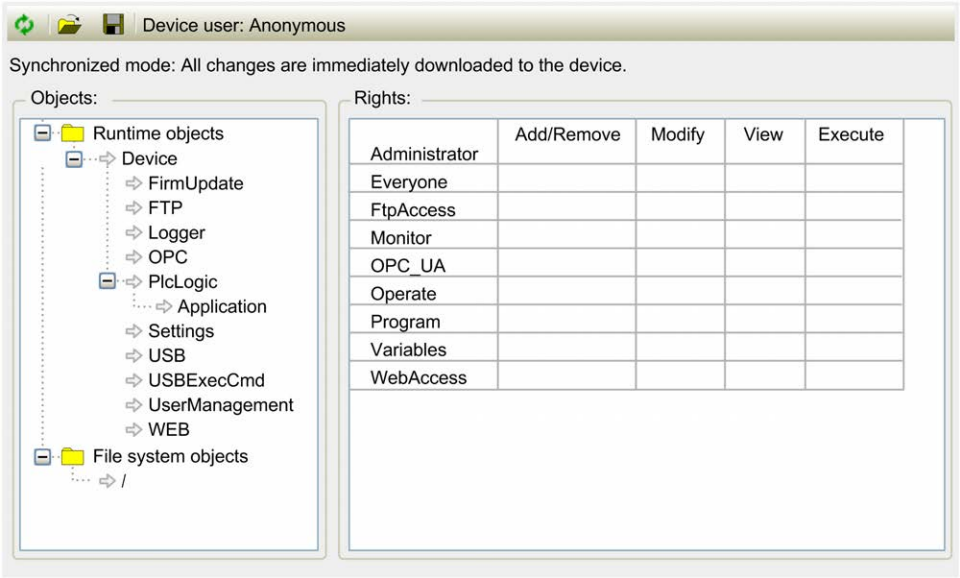
Access Rights

Overview

In the **Access Rights** view of the device editor, define the device access rights of device users to objects in the controller.

In order for the **Access Rights** view to be available in the device editor of controllers, activate the **Show access rights page** option in the **Tools > Options > Device Editor** dialog box (see Menu Commands, Online Help). Furthermore, **Users and Groups** management must be set up in the controller.

Example of an **Access Rights** view of the device editor



NOTE: Detailed information on the concept and use of device user management is provided in *Handling of Device User Management* in the *Security* chapter, page 167.

Modifying Access Rights to Controller Objects in the Users and Groups Management of the Controller

If the controller supports device **Users and Groups** management, you can assign access rights as follows:

Step	Action	Comment
1	Double-click the controller node in the **Devices tree**.	**Result**: The device editor opens.
2	Select the **Access Rights** view.	–
3	Click the **Synchronization** button  to load the **Access Rights** management configuration from the controller to the editor.	If you are not logged in to the controller yet, then the dialog box **Device User Login** opens. It allows you to enter the user name and the password. **Result**: The **Access Rights** management configuration of the controller is displayed in the editor.
4	In the **Objects** tree structure on the left-hand side, select the object.	**Result**: In the **Rights** area on the right-hand side, the access rights of the selected object are displayed in a table for the configured user groups.
5	In the **Rights** table, double-click the access right you want to modify.	If the selected object has child objects, a dialog box is displayed, prompting you whether you want to modify the access rights for the child objects at the same time.
6	Click **Yes** or **No** to modify the access rights of the child objects and to close the dialog box.	**Result**: The access rights are switched from allowed to not allowed or vice versa. The symbol in the table cell is modified accordingly. Rights that are set explicitly are displayed in the table by green or red symbols. Rights that are inherited from a parent object are displayed in the table by gray symbols.

Toolbar of the Access Rights View

The toolbar provides the following elements:

Element	Description
Synchronization	Click the **Synchronization** button to switch on / off the synchronization between the editor and the **Access Rights** management in the controller. If **Synchronization** is not activated, then the editor contains an **Access Rights** management configuration that has been imported from disk, or it does not contain any configuration. If **Synchronization** is activated, the data displayed in the editor is continuously synchronized with the **Access Rights** management configuration on the connected controller. If you invoke **Synchronization** while the editor contains an **Access Rights** management configuration that is not synchronized with the device, you are prompted to decide what will be displayed in the editor: • **Upload from the device and overwrite the editor content**: The **Access Rights** configuration from the controller is loaded to the editor. The contents of the editor is overwritten. • **Download the editor content to the device and overwrite the user management there**: The configuration from the editor is loaded to the controller. The contents of the controller is overwritten.
Load from disk	When you click the **Load from disk** button in the **Access Rights** view, the file type is by default set to *Device rights management* files (*.drm). The existing configuration will be overwritten by the imported file.
Save to disk	When you click the **Save to disk** button in the **Access Rights** view, the file type is by default set to *Device rights management* files (*.drm). For this file, a password does not have to be assigned before saving.
Device user	Name of the user who is logged into the controller.

Objects Area

In the **Objects** tree structure on the left-hand side, the objects are listed that allow actions to be executed in runtime mode. The objects are assigned by their object source. They are partially sorted in object groups. In the **Rights** area on the right-hand side, you can configure the access options of the selected object for a user group.

On the top level of the **Objects** tree structure there are two object categories grouped in folders:

- **Runtime objects**
- **File system objects**

Indented below the object categories, there are further subnodes. The subnode **Device**, for example, can, in turn, have the following subnodes:

- **Logger**
- **PlcLogic**
- **Settings**
- **UserManagement**

A description of the objects is provided in the paragraph Overview of the Objects, page 141.

Rights Area

In general, access rights are inherited from the root node (**Device** or **/**) to the subnodes. If a permission of a user group is denied or explicitly granted to a parent object, then this is also applied to the child objects.

In the **Rights** area on the right-hand side, the access rights of the selected object are displayed in a table. For every user group, it displays the rights configured for the possible actions on the selected object.

The following actions can be configured for the object:

- **Add/Remove**
- **Modify**
- **View**
- **Execute**

The symbols represent the access rights:

Icon	Description
	Access (action) is permitted explicitly.
	Access (action) is denied explicitly.
 	The access right has been inherited from the parent object.
	Access has not been permitted or denied explicitly, even for the parent object. No access is possible
No icon	Several objects with different access rights are selected.

To modify an access right, click the symbol.

Example

The **Logger** node in the **Access Rights** tab is created by the logger component and controls its access rights. It is located directly below the **Runtime objects > Device** node.

For this object, you can only grant **View** access rights.

By default, each object is assigned read access. Thus, every user can read the logger of a controller.

To deny this access right for a single user group (**Service**, for example), set the **View** right for the **Logger** object to .

Printing the Access Rights Definition

To print the settings of the **Access Rights** view, run the command **Print** from the **File** menu or the command **Document** from the **Project** menu.

Overview of the Objects

The following table provides an overview of general runtime objects. Consult the *Programming Guide* specific to your controller for controller-specific objects.

Object	Description
Runtime objects > Device	
Logger	Online access to the logger is read only. Therefore, only the **View** access right can be granted or denied.
PlcLogic	Applications are inserted as child objects during download. If applications are deleted, the child object subnodes are also removed here. This allows for dedicated control of online access to the application. The **PlcLogic** object allows you to assign access rights at a central place for all applications: • The **Administrator** and **Developer** user groups have full access to the applications. • The **Service** and **Watch** user groups have read access (for example, for read-only monitoring of values). Also refer to the table in the *Access Rights for IEC Applications* paragraph, page 142.
RemoteConnections	The **RemoteConnections** object allows you to configure additional external connections to the controller, such as access to the OPC UA server.
Settings	Online access to the configuration settings of a controller. By default, the access right **Modify** is granted to the administrator exclusively.
UserManagement	Online access to the user management of a controller. By default, the access right **Modify** is granted to the administrator exclusively.
X509	Online access to the X.509 certificates: • Read (**View**) • Write (**Modify**) Each operation is inserted as a child object below **X509** and one of the two access rights is assigned. This allows you to specify the access per operation.
FTP	Allows you to configure the access rights to the FTP server on the controller.
OPC	Allows you to configure the access rights to the controller using OPC UA.
USB	Allows you to configure the access rights to files on a USB storage device connected to the controller.
USBExecCmd	Allows you to configure the access rights for executing scripts located on a USB storage device connected to the controller.
WEB	Allows you to configure the access rights to the webserver on the controller.
File system objects > /	
–	The folders of the execution path of the controller are inserted below the **/** file system object. This allows you to grant specific rights to each folder of the file system.

Access Rights for Applications

The table indicates which action is affected when a specific access right is granted for an **Application**:

Operation	Access rights			
	Add/ Remove	Execute	Modify	View
Login	–	–	–	X
Create	X	–	–	–
Create child object	X	–	–	–
Delete	X	–	–	–
Download / online change	X	–	–	–
Create boot application	X	–	–	–
Read variable	–	–	–	X
Write variable	–	–	X	X
Force variable	–	–	X	X
Set and delete breakpoint	–	X	X	–
Set next statement	–	X	X	–
Read call stack	–	–	–	X
Single cycle	–	X	–	–
Switch on flow control	–	X	X	–
Start / Stop	–	X	–	–
Reset	–	X	–	–
Restore retain variables	–	X	–	–
Save retain variables	–	–	–	X
(X) The access right must be set explicitly. **(–)** The access right is not relevant.				

Symbol Rights

Overview

The **Symbol Rights** view of the device editor is used to configure the access rights of different user groups (clients) to the individual symbol sets, page 514 available on the controller.

The following prerequisites must be fulfilled:

- User management must be enabled and configured on the controller.
- An application has been downloaded to the controller for which symbol sets were defined in the project.
- You have the required login data to gain access to the controller.

Description of the Views

The following views are available:

View	Description
Symbol Sets	The **Application** node contains the symbol sets whose definition was downloaded with the application to the controller.
Rights	The table contains the user groups defined in the user management of the controller. When a symbol set is selected, the access rights of the corresponding user group to the symbols of this set are indicated: • : Access granted. • : Access not granted. You can change the access rights by double-clicking the symbol.

Click the **Save to disk** button to save the access configuration to an XML file of type *Device symbol management* with the extension *.dsm.

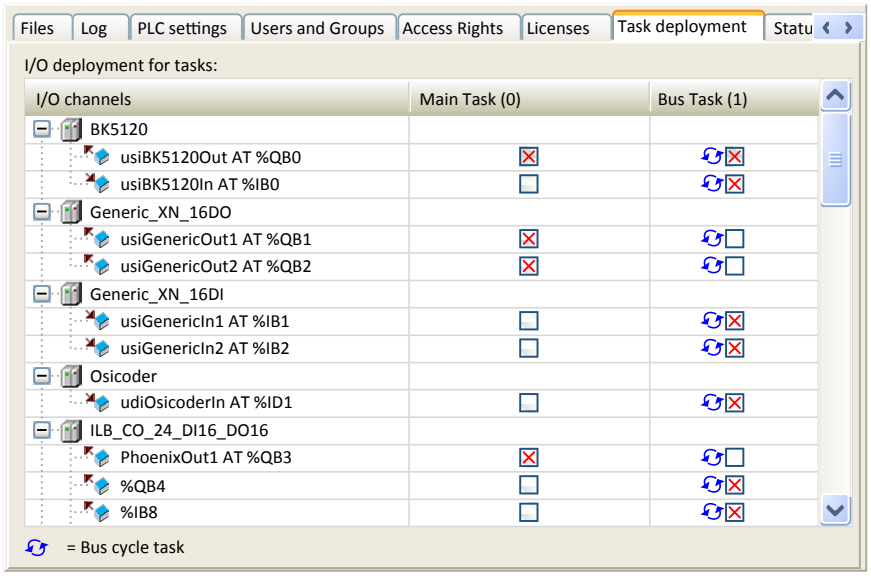
Click the **Load from disk** button to load a *.dsm file from the hard drive.

Task Deployment

Overview

The **Task deployment** view of the device editor shows a table with inputs/outputs and their assignment to the defined tasks. Before the information can be displayed, the project has to be compiled and the code has to be generated. This information helps in troubleshooting in case that the same input/output is updated in different tasks with different priorities.

Task deployment of the device editor



The table shows the tasks sorted by their task priority. Click the column heading (**Main Task**) to display only the variables assigned to this task. To show all variables again, click the first column (**I/O channels**).

To open the I/O mapping table of a channel, double-click the input or output.

A blue arrow indicates the task of the bus cycle.

In the example above, the variable **usiBK5120Out AT %QB0** is used in 2 different tasks. In this situation, the output, set by one task, can be overwritten by the other task: this can lead to an undefined value. Writing output references in more than one task makes the program difficult to debug and may lead to unintended results in the operation of your machine or process.

⚠WARNING

UNINTENDED EQUIPMENT OPERATION

Do not write to an output variable in more than one task.

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

Status

Overview

The **Status** view of the device editor shows status information (for example, **Running**, **Stopped**) and specific diagnostic messages from the device; also on the card and the internal bus system.

Information

Overview

The **Information** view of the device editor shows some general information on the device selected in the **Devices tree**: **Name, Vendor, Type, Version number, Order Number, Description, Image**.

Emulation Settings

Overview

The **Emulation Settings** view of the device editor is available for PacDrive LMC controllers. It is used to configure the emulation with **EcoStruxure Machine Expert Twin** running in the background and the communication between **EcoStruxure Machine Expert Twin** and the controller.

For further information, refer to the How to Emulate User Guide.

Description of the Elements

Element	Description
Machine Expert Twin	
Open User Interface	• If this option is activated, the user interface of **EcoStruxure Machine Expert Twin** opens when an emulation is started. After the session has been established, the configured object, for example, one multi carrier track or one robot, is visible in the **EcoStruxure Machine Expert Twin** user interface. The **Emulation** tab of the **Multicarrier Configuration** editor returns to its initial state. • If this option is not activated, the **EcoStruxure Machine Expert Twin** opens in the background when an emulation is started. After the session has been established, the configured object is only visible in the tool you are using, for example, the **Emulation** tab of the **Multicarrier Configuration** editor.
<n> sec - Machine Expert Twin startup timeout	You can configure a timer (in seconds) that specifies the time to wait for the startup of **EcoStruxure Machine Expert Twin** in the background before an error message is displayed. If the default setting of 25 seconds is insufficient, increase the timer value. Alternatively, you can start **EcoStruxure Machine Expert Twin** separately in standalone mode to reduce the loading time.
<n> ms - Communication Interval	You can configure the data exchange interval between **EcoStruxure Machine Expert Twin** and the controller over OPC UA. Increasing the value reduces the CPU load on the controller but can reduce the quality of the visual representation of the emulation. Default value: 16 ms
OPCUA Server on Controller	
IP Address from Controller	• If this option is activated, the field **IP** is disabled and the IP address of the OPC UA server is retrieved from the controller configuration. **NOTE:** To use this option, the **Connection Mode** > **IP Address** must be selected in the **Communication Settings** tab of the device editor, page 101. • If this option is not activated, enter the IP address of the OPC UA server in the field **IP**.
Anonymous Login	• If this option is activated, user name and password are not required to establish a connection to the OPC UA server. **NOTE:** Activate this option only in case the option **Disable anonymous login** was not activated in the **General Settings** tab of the **OPC UA Server Configuration** of the PacDrive LMC controller. • If this option is not activated, you are requested to enter user name and password when you start the emulation to establish a connection to the OPC UA server. **NOTE:** The credentials are retained until the project is closed.

I/O Mapping

I/O Mapping

Overview

The **I/O Mapping** view of the device editor is named **<devicetype> I/O Mapping** (for example, **PROFIBUS DP I/O Mapping**). It serves to configure an I/O mapping of the controller. Therefore, project variables used by the application are assigned to the input, output, and memory addresses of the controller.

Define the application which should handle the I/Os in the **PLC settings** view, page 128.

See the following chapters:

- Working with the I/O Mapping Dialog, page 148
- I/O Mapping in Online Mode, page 154
- Implicit Variables for Forcing I/Os, page 154

General Information on Mapping I/Os on Variables

Whether an I/O mapping can be configured for the present device depends on the device. It can be that the view is only used to show the implicitly created device instance. See description of the *IEC objects*, page 130.

Basically, note the following for the mapping of I/Os to variables:

- Variables requiring an input cannot be accessed by writing.
- An existing variable can only be mapped to one input.
- Instead of using the **I/O Mapping** view, you can also assign an address to a variable using the AT declaration, page 562.

  However, consider the following:

  - You can use AT declarations only with local or global variables; not however, with input and output variables of POUs.
  - The possibility of generating force variables for I/Os (refer to Implicit Variables for Forcing I/Os, page 154) will not be available for AT declarations.
  - If AT declarations are used with structure or function block members, all instances will access the same memory location. This memory location corresponds to static variables in classic programming languages such as C.
- The memory layout of structures is determined by the target device.
- For each variable which is assigned to an I/O channel in the **I/O Mapping** view, force variables can be created during a build run of the application (see Menu Commands, Online Help). You can use them for forcing the input or output value during the commissioning of a machine, for example, trough a visualization. Refer to the chapter *Implicit Variables for Forcing I/Os*, page 154.

Automatic I/O Mapping

The automatic I/O mapping function automatically creates IEC variables as soon as a device or module with I/O modules is added to the **Devices Tree** and maps them on each input and/or output. By default, the function is activated.

You can deactivate and configure the function in the **Project > Project Settings > Automatic I/O mapping** dialog box.

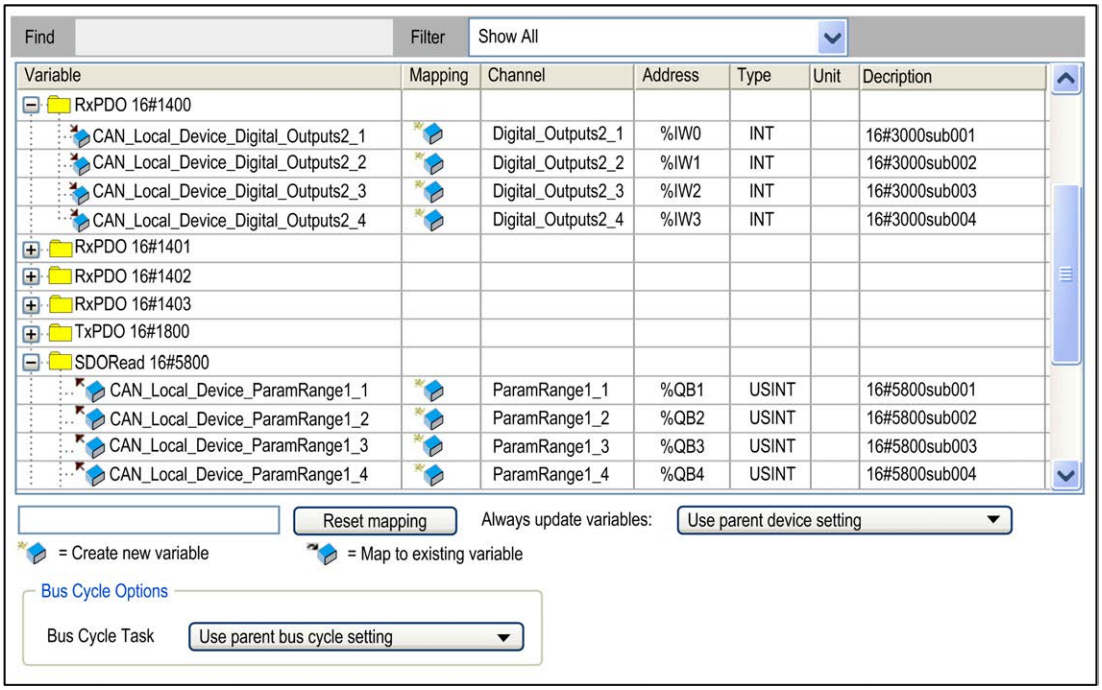
The dialog box provides the following elements:

Element	Description
Create variable on I/Os mapping	By default, the option is activated. Select this option to activate the automatic I/O mapping function and to assign names of your choice to the inputs and outputs.
Mapping area	
Bitwise	Select this option to create variables for each bit.
Module-wise	Select this option to create a variable for each module, not for the individual bits.
Naming Rule area	
text box	Enter the following characters preceded by a `#` symbol to specify the parts the variable name will consist of: • Enter `#X` to integrate an `i` for inputs and a `q` for outputs in the variable name. • Enter `#T` to integrate the prefix code for the data type of the variable in the variable name. The prefixes that are used for the different data types are listed in the *Best Practices for the Naming of Identifiers* chapter, page 553. • Enter `#D` to integrate the name of the device in the variable name. • Enter `#C` to integrate the name as defined in the **Channel** column in the variable name.

Working with the I/O Mapping Dialog Box

Overview

The following is an illustration of the **I/O Mapping** tab of the device editor:



Description of the Elements in the Channels Area

The **I/O Mapping** tab provides the following elements in the **Channels** area if provided by the device:

Element	Description
Channel	Symbolic name of the input or output channel of the device
Address	Address of the channel, for example: `%IW0`
Type	Data type of the input or output channel, for example: BOOL If the data type is not standard, but a structure or bit field defined in the device description, it will be listed only if it is part of the IEC 61131–3 standard. It is indicated as IEC type in the device description. Otherwise, the entry of the table will be empty.
Default value	This column is only available if the option **Set all outputs to default** is selected for the parameter **Behaviour for outputs in Stop** in the **PLC Settings** view of the device editor, page 128. Default value that is assigned to the channel when the controller is set to STOP mode. You can edit this field only if you are mapping to a new created variable or if no mapping is specified. When you are mapping to an existing variable, the initialization value of the variable is used as the default value. **NOTE:** In case a “new” variable and an “existing” variable (by using the `AT` declaration) are mapped to the same output, the initialization value of the “existing” variable is used as default value. **NOTE:** You can modify the default value using an online change. The new value will be applied when executing a **Reset cold** or **Reset warm**.
Unit	Unit of the parameter value, for example: **ms** for milliseconds
Description	Short description of the parameter
Current Value	This column is available only in online mode. Present value of the parameter.

NOTE: Inputs and outputs that are not used in the application are not read by the controller in online mode. To indicate that these inputs and outputs are not used, they are marked with a gray background. Any values that might appear in these gray lines are invalid.

⚠WARNING

UNINTENDED EQUIPMENT OPERATION

Do not map other user variables in the **I/O Mapping** tab if you are using libraries for fieldbus communications that are reading/writing from/to direct addresses (`%I`, `%Q`).

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

Consult the documentation of your fieldbus library to see if direct addresses are used.

Modifying and Locking Addresses

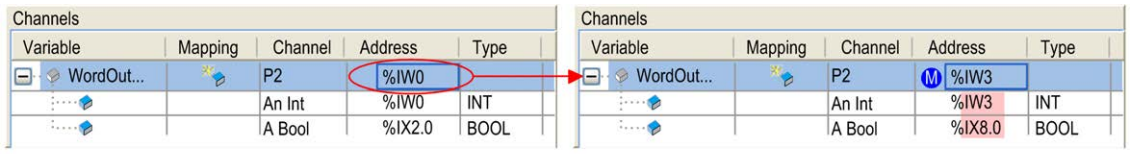
This function is not available for all supported controllers. Consult the *Programming Guide* specific to your controller for further information.

You can modify and lock the displayed address of an output or input here in this tab. Use this to adapt the addressing to a given hardware configuration or to keep the address value even if the order of the modules is changed. By default, this would cause an automatic adaptation of the address values.

Consider that depending on the device description, you can only modify the address of the input or output, however, not that of its subelements (bit channels). Therefore, if an input or output is represented here in the mapping table with a subtree, you can edit only the address field of the uppermost entry (see the figure below: only the address field in the first line can be opened).

In order to fix the address value, select the entry in the **Address** column and press the SPACE bar to open the edit field. Either modify the value or leave it unmodified and close the edit field via the RETURN key. The address field is marked by an **M** symbol which indicates that the value has been modified.

If the value has been modified, the subsequent addresses (up to the next address) will be adapted correspondingly:



If you want to revert the modification of the value, reopen the address edit field, delete the address entry, and close with **Enter**. The address and the identified succeeding addresses will be set back to the values they had before the manual modification. The **M** symbol will be removed.

Configuration of the I/O Mapping

Perform the I/O mapping by assigning the corresponding project variables to the device input and output channels each in the **Variable** column.

- The type of the channel is already indicated in the **Variable** column by a symbol: for input, for output. In this line, enter the name or path of the variable to which the channel should be mapped. You can either map on an existing project variable or define a new variable, which then will automatically be declared as a global variable.
- When mapping structured variables to outputs, the editor will prevent that both the structure variable (for example, on `%QB0`) and particular structure elements (for example, in this case on `%QB0.1` and `QB0.2`) can be entered.

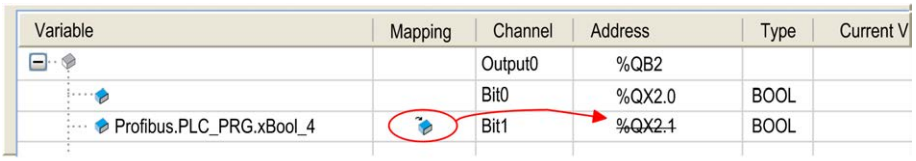
  This means: When there is a main output entry with a subtree of bit channel entries in the mapping table, then either in the line of the main entry a variable can be entered or in those of the subelements (bit channels) never in both.
- For mapping on an existing variable, specify the complete path. For example: <application name>.<pou path>.<variable name>';

  Example: `app1.plc_prg.ivar`

  For this purpose, it can be helpful to open the input assistant via the **...** button.

  In the **Mapping** column, the symbol will be displayed and the address value will be crossed out. This does not mean that this memory address does not exist any longer. However, it is not used directly because the value of the existing variable is managed on another memory location, and, especially in case of outputs, no other already existing variable should be stored to this address (`%Qxx` in the I/O mapping) in order to avoid ambiguities during writing the values.

See in the following example an output mapping on the existing variable `xBool_4`:



NOTE: When you are mapping to an existing variable, the initialization value of the variable is used as the default value. You can edit the **Default value** field only if you are mapping to a new created variable or if no mapping is specified.

- If you want to define a new variable, enter the desired variable name.

  Example: `bVar1`

  In this case, the symbol will be inserted in the **Mapping** column and the variable will be internally declared as a global variable. From here, the variable will be available globally within the application. The mapping dialog box is another place for the declaration of global variables.

  NOTE: Alternatively, an address can also be read or written within a program code, such as in ST (structured text).
- Considering the possibility of changes in the device configuration, do the mappings within the device configuration dialog box.

  NOTE: If a UNION is represented by I/O channels in the mapping dialog box, it depends on the device whether the root element is mappable or not.

If a declared variable of a given data type is larger than that to which it is being mapped, the value of the variable being mapped will be assigned a truncated to the size of the mapped target variable.

For example, if the variable is declared as a WORD data type, and it is mapped to a BYTE, only 8 bits of the word will be mapped to the byte.

This implies that, for the monitoring of the value in the mapping dialog box, the value displayed at the root element of the address will be the value of the declared variable - as currently valid in the project. In the subelements below the root, the

particular element values of the mapped variable will be monitored. However, only part of the declared value may be displayed among the subelements.

A further implication is when you map a declared variable to physical outputs. Likewise, if you map a data type that is larger than the output data type, the output data type may receive a truncated value such that it may affect your application in unintended ways.

⚠WARNING

UNINTENDED EQUIPMENT OPERATION

Verify that the declared data type that is being mapped to physical I/O is compatible with the intended operation of your machine.

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

Element	Description
Reset mapping	Click this button to reset the mapping settings to the defaults defined by the device description file.
Always update variables	Definition if the I/O variables are updated in the bus cycle task, page 128. The default value is defined in the device description. • **Use parent device settings**: Update according to the settings of the parent device. • **Enabled 1 (use bus cycle task if not used in any task)**: The I/O variables are updated in the bus cycle task if not used in another task. • **Enabled 2 (always in bus cycle task)**: The variables are updated in every cycle of bus cycle task, regardless of whether they are being used or whether they are mapped to an input or to an output channel.

Bus Cycle Options

This configuration option is available for devices with cyclic calls before and after reading inputs or outputs. It allows you to set a device-specific bus cycle task, page 128.

Per default, the parent bus cycle setting will be valid (**Use parent bus cycle setting**). Therefore, the **Devices Tree** will be searched for the next valid bus cycle task definition.

To assign a specific bus cycle task, select the desired one from the selection list. The list provides the tasks currently defined in the application task configuration.

⚠WARNING

UNINTENDED EQUIPMENT OPERATION

Do not write to an output variable in more than one task.

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

Mapping a Device and a Function Block Instance

Overview

If supported by the device, you can map function blocks to an input or output channel. This allows you, for example, to count the frequency of signal changes or scale a channel value for maintenance purposes.

The following code example maps a device output channel to a function block. The function block scales the channel output value.

The following prerequisites must apply:

- A device with an analog output of type INT that supports function block mapping is available in the project.
- A function block *Scale_Output_Int* with the following implementation is available.

Consider the attributes of the function block itself and the attributes preceding the output parameter for processing the channel output.

```
{attribute 'io_function_block'}
FUNCTION_BLOCK Scale_Output_Int
VAR_INPUT
    iInput : INT;
    iNumerator : INT;
    iDenominator : INT :=1;
    iOffset : INT := 0;
END_VAR
VAR_OUTPUT
 {attribute 'io_function_block_mapping'}
    iOutput : INT;
END_VAR
VAR
END_VAR
IF iDenominator <> 0 THEN
    iOutput := TO_INT(TO_DINT(iInput) * TO_DINT(iNumerator)
/ TO_DINT(iDenominator)) + iOffset;
```

Procedure

To map a device and a function block instance, proceed as follows:

Step	Action	Comment
1	Open the **<device name> I/O Mapping** tab of the device editor.	–
2	Select the output you want to map to the function block.	–
3	Click the plus button .	**Result**: The **Select Function Block** dialog box opens. The left-hand side displays at least the function block *Scale_Output_int* below the **Application** node. Libraries available in the project that contain the corresponding function blocks are also displayed for selection.
4	Select the function block *Scale_Output_int* and click **OK**.	**Result**: The path of the function block parameter *iOutput* is entered in the column **Variable** of the I/O mapping table.
5	Select the channel and click the **Go to Instance** button.	**Result**: The focus switches to the **<device name> IEC Objects** view, page 130 and to the new entry. The **IEC Objects** view in online mode displays the function block. You can write (see Menu Commands, Online Help) and force (see Menu Commands, Online Help) the value in this monitoring view.

I/O Mapping in Online Mode

I/O Mapping in Online Mode

If a structure variable is mapped on the root element of the address (the uppermost in the tree of the respective address in the mapping dialog box), then in online mode no value will be displayed in this line. If, however, for example, a DWORD variable is mapped to this address, then in the root line, as well as in the bit channel lines indented below, the respective values will be monitored. Basically, the field in the root line stays empty if the value is composed of multiple subelements.

Implicit Variables for Forcing I/Os

Overview

During the commissioning of a plant or a machine, it may be necessary to force I/Os. For this purpose, you can generate special force variables for each I/O channel which is mapped on a variable in the **I/O Mapping** tab of the device editor.

As a precondition the setting **Generate force variables for IO mapping** has to be activated in the **PLC settings** tab. Then, at each build run of the application, for each mapped I/O channel, two variables are generated according to the following syntax. Empty spaces in the channel name are replaced by underscores.

<devicename>_<channelname>_<IECaddress>_Force of type BOOL, for activating and deactivating the forcing

<devicename>_<channelname>_<ECaddress>_Value of datatype of the channel, for defining the value to be forced on the channel

These variables are available in the input assistant in category **Variables > IoConfig_Globals_Force_Variables**. They can be used in programming objects, in visualizations, symbol configuration, and so on, within the programming system.

A rising edge at the force variable activates the forcing of the respective I/O with the value defined by the value variable. A falling edge deactivates the forcing. Deactivating by setting the force variable back to FALSE is necessary before a new value can be forced.

Consider the restrictions listed below.

Example

If the mapping is completed as shown in figure **I/O Mapping** tab of the device editor, page 148, then at a build (F11) of the application, the following variables are generated and available in the input assistant:

- `Digitax_ST_Control_word_QW0_Force : BOOL;`
- `Digitax_ST_Control_word_QW0_Value : UINT;`
- `Digitax_ST_Target_position_QD1_Force : BOOL;`
- `Digitax_ST_Target_position_QD1_Value : DINT;`
- `Digitax_ST_Status_word_IW0_Force : BOOL;`
- `Digitax_ST_Status_word_IW0_Value : UINT;`
- `Digitax_ST_Position_actual_value_ID1_Force : BOOL;`
- `Digitax_ST_Position_actual_value ID1_Value : DINT;`

Restrictions

- Only channels which are mapped on a variable in the **I/O Mapping** tab (i.e., a variable has to be defined in the **Variable** column, no matter whether it is a new or an existing one) can be forced by the above described implicit variables.
- Unused inputs / outputs as well as those which are mapped via AT declaration in an application program cannot be forced.
- The respective I/O channels have to be used in at least one task.
- Forced I/Os are not indicated in the monitoring (watch view, I/O mapping dialog box). The value is only used implicitly in the I/O driver for writing onto the device.
- Forced inputs are displayed correctly by the red force symbol (**F**), not however, forced inputs/outputs.

Select Function Block Dialog Box

Overview

The **Select Function Block** dialog box allows you to select a function block for I/O mapping. If the function block is mapped to the I/O channel selected in the **I/O Mapping** or the **IEC Objects** view, click the plus button to open the dialog box.

The dialog box lists the function blocks of the active application and the libraries included in the project that meet the following requirements:

- The attribute `{attribute 'io_function_block'}` is assigned to the function block.

- The attribute `{attribute 'io_function_block_mapping'}` is assigned to input or output parameters of the function block that match the channel type (input, output, data type).

When you select a function block that contains more than one matching parameters, the first parameter is mapped to the channel. To assign further parameters manually, open the **I/O Mapping** tab.

After a function block has been mapped, the parameter of the function block instance is added to the **Variable** column of the mapping table. The path consists of the following elements:
`<application name>.<device channel name>_<FB name>_ <continuous FB instance number>. <FB parameter name>`

Example:

`App1.Out_4_Int_myScale_Output_Int_1.iOutput` for the parameter `iOutput` of the first inserted instance of the function block *myScale_Output*.

Elements of the **Select Function Block** dialog box:

Element	Description
Find	Text field for entering function block name to search for.
Type	Function blocks in the tree structure that match the channel type. The following nodes are available: application, library name(s).
Documentation	Displays the documentation available for the library or the function block of the library selected in the tree structure.

Cybersecurity

What's in This Part

General Information 158
Digital Signature Verification 159
Connecting to the Controller 162
Logoff Current Device User 163
Security for the Project and the Application 164
Security for the Runtime System / Controller 167
Visualization 170
FAQ 171

Overview

Cybersecurity best practices and solutions are in constant evolution as a function of the latest information available. As a design criteria, Schneider Electric incorporates up-to-date knowledge and techniques to help make products more resilient to cyberattacks. The security by design approach results in the implementation of mechanisms to mitigate threats, reduce exploitable weaknesses, and defend against avoidable data breaches and cyberattacks.

> **NOTE:** To help keep your Schneider Electric products secure and protected, it is in your best interest that you implement the cybersecurity best practices as indicated in the *Cybersecurity Best Practices* document provided on the Schneider Electric website.

Due to the rapid rise of networking machines and plants, potential threats are also quickly rising. Therefore, you must carefully consider all possible security measures.

Security measures are necessary to help protect data and communication channels from unauthorized access.

This part provides information about the cybersecurity mechanisms provided by EcoStruxure Machine Expert and the controllers.

General Information

What's in This Chapter

Access Protection with Device User Management 158
Firewall Settings 158
Using Certificates 158

Access Protection with Device User Management

Introduction

In order to meet constantly evolving cybersecurity requirements, the user rights management for EcoStruxure Machine Expert controllers is by default activated. Therefore, every EcoStruxure Machine Expert controller equipped with the latest firmware prompts you for user credentials whenever you attempt to gain access to it.

For general information regarding device user management, refer to the chapter describing the **Users and Groups** view of the device editor, page 132.

Firewall Settings

Firewall Settings on the Controller for Communication Services

Most of the communication services like FTP or OPC UA access the controller by using the settings of the user rights management. Therefore, make sure that the firewall settings on the controller allow the services to access the controller file system. For further information, refer to “How to Configure the Firewall...” for your specific controller.

Using Certificates

Overview

A certificate is a digital representation of information which contains the following:

- Identifies the certification authority issuing it.
- Names or identifies the subscriber.
- Contains the public key of the subscriber.
- Identifies its operational period.
- Is digitally signed by the certification authority issuing it.

Optionally, a certificate can be created for a specific usage. For example, you cannot use a controller certificate for protecting a project and vice versa.

The certificate consists of two parts / files:

- Public X.509 certificate (can be issued to anyone).
- Private key that matches the certificate or its public key only (must be kept secret).

Digital Signature Verification

What's in This Chapter

Digital Signature Verification 159

Digital Signature Verification

Verifying the Integrity of Your Software

When your software (such as EcoStruxure Machine Expert) is started, each loaded dynamic-link library (DLL) is scanned to verify whether or not it is trusted. This is a built-in security feature to help increase the trust level.

Reporting Untrusted Components

If untrusted DLLs are detected, you are informed depending on the verification level:

- In the status bar.
- In the dialog box **Integrity Verification**.
- In the console, such as Logic Builder Shell.

The result of the verification is documented in the log file of your software.

Verification Levels

The table lists the verification levels and the behavior in the software depending on whether it is started with our without user interface:

Verification Level	Software started with user interface	Software started without user interface
`Error` (default)	The dialog box **Integrity Verification** is displayed (blocking start-up of your software). You have the choice to continue or terminate the process.	The error message is written to the console, such as Logic Builder Shell, and to the log file of your software. The process is terminated.
`Warning`	A notification is displayed in the status bar and is documented in the log file of your software. The process continues.	A notification is written to the console, such as Logic Builder Shell, and to the log file of your software. The process continues.
`Silent`		The notification is documented in the log file of your software.

The verification level is applied per process and is not set globally. Once the verification level is set, it cannot be changed for a running software process.

Configuring the Verification Level

Different options are provided for configuring the verification level.

- As the preferred option, edit the value of the `SignatureVerificationWarningLevel` in the **.exe.config* configuration file of your Schneider Electric application that is available in the installation directory (*SchneiderElectric.<NAMEOFYOURAPPLICATION>.exe.config*).

```
<?xml version="1.0"?>
<configuration>
    <appSettings>
        <!-- Default SignatureVerificationWarningLevel [Error, Warning, Silent].
It can be overriden by command line arguments. -->
        <add key="SignatureVerificationWarningLevel" value="Error"/>
    </appSettings>
</configuration>
```

  By default, the `SignatureVerificationWarningLevel` is set to the value `Error`.

- With the command line argument `"-- SignatureVerificationWarningLevel=[Error|Warning|Silent]"`

  To specify the verification level argument, run your software from console or adapt the **Start** menu shortcut.

  NOTE: The verification level set with the command line argument overrides the value configured in the configuration file of the application.

Verification Log Information

The digital signature verification results can be exported to the **Verification Log** (see dialog box **Integrity Verification**).

Untrusted digital signature verification results are written to:

- the Windows EventLog (`Source: "Schneider Electric Software"`).
- the log file of your software, such as EcoStruxure Machine Expert (see the *\TEMP* directory).

How to Proceed Depending on the Verification Level

Verification level = `Error`:

If the verification level `Error` applies, the dialog box **Integrity Verification** is displayed when your software is started. It provides the following options:

- The **Export Verification Log...** button allows you to save the log information to an XML file. Consult this file to get more information about the DLLs that were detected as untrusted.
- The **Exit** button allows you to exit the software promptly.
- To continue using your software, select the check box indicating that you have taken the notification about untrusted components into account and click the button **Continue**. Note that the notification will appear with every start of your software.

Verification level = `Warning`:

If the verification level `Warning` applies, the startup process of your software continues and a notification is displayed in the statusbar. Clicking this notification opens the **Integrity Verification** dialog box which allows you to consult the **Verification Log** for more information about the DLLs that were detected as untrusted.

Analyze each notification before you continue to work with the software. As there are many reasons for untrusted components, there are also different ways to proceed:

- If an expected reason is reported, you can continue to work with the software.
- If an unexpected reason triggered by an installed third-party product on the same PC is reported, you can continue to work with the software.
- If, however, an untrusted component is reported for an unexpected reason and for which you do not recognize the component, analyze the **Verification Log** to find out why a DLL was detected as untrusted before you continue to work with the software.

Connecting to the Controller

What's in This Chapter

First Login to the Controller with User Rights Management Activated 162

First Login to the Controller with User Rights Management Activated

Procedure

The following login procedure applies to controllers running firmware released with EcoStruxure Machine Expert V2.0 and later versions. For the procedure applicable to EcoStruxure Machine Expert V1.2.• versions, refer to the corresponding Release Notes.

NOTE: The following procedure is only allowed for users with access rights for **Administrator** and **Program**.

Step	Action
1	Enter a new user name.
2	Enter your individual **Password**. **NOTE:** The password must comply with the password policy configured in the **Change Runtime Password Policy** dialog box, page 116.
3	Re-enter your individual **Password**.
4	Click **OK** to confirm. **Result**: You are requested to enter the new credentials for accessing the controller.

NOTE: For future login, the new **Password** will be required.

Logoff Current Device User

What's in This Chapter

Logoff / Change Device User 163

Logoff / Change Device User

Procedure

After successful login to the controller, you can perform further online actions on the controller with EcoStruxure Machine Expert. As long as your project remains open, you will not be prompted to enter your credentials again.

In order to log off or change the present user from the controller, run the **Online > Security > Logoff current device user** command (see Menu Commands, Online Help).

After that you will be prompted for your credentials when you attempt to perform another online command on the controller.

Security for the Project and the Application

What's in This Chapter

Protecting and Saving a Project (exclusive to *.project and *.library files) 165
Protecting and Saving Applications 165

NOTE: The protection of a file-based project (*.fbs) relies on the access to the project folder. Since the project files are in a human-readable format, allow access to authorized persons only.

Protecting and Saving a Project (exclusive to *.project and *.library files)

Overview

This function is exclusive to *.project and *.library files in classic storage format.

EcoStruxure Machine Expert allows you to apply access protection to projects, libraries, as well as individual applications. In addition to write-protecting a project, it is a good practice to implement a user management (credentials, access rights) and encryption using certificates.

- The user management offers protection for the end user against accidental changes to the protected objects. The **Project Settings > Users and Groups** dialog box (see Menu Commands, Online Help) is used to manage user accounts and user access rights groups and the associated access rights. This allows you to control the access on EcoStruxure Machine Expert projects.
- The **Project > User Management > Permissions...** command (see Menu Commands, Online Help) opens the **Permissions** dialog box for configuring the rights to work on objects to run commands in the open project or to modify the user management configuration.
- The user management does not serve as an Intellectual Property Protection (IPP) of the function blocks. EcoStruxure Machine Expert itself, as well as plug-ins and persons with knowledge of the project file format can view or change function blocks.

  NOTE: Intellectual Property Protection of libraries can be enhanced when libraries are delivered as compiled libraries. Therefore, run the **File > Save Project As Compiled Library** command (see Menu Commands, Online Help) to save a library in encoded format. This library no longer contains source code. Only the compiler is able to interpret these data; other EcoStruxure Machine Expert components are denied this information.
- Intellectual Property Protection of projects can be greatly enhanced by encrypting the project file. To achieve this, activate the option **Enable project file encryption** in the **Project Settings > Security** dialog box (see Menu Commands, Online Help), and select the encryption option that suits your needs:
  - Protection with Password (see Menu Commands, Online Help)
  - Protection with Dongle (see Menu Commands, Online Help)
  - Protection with Certificates (see How To Help Protect Your Source Code, User Guide)

  NOTE: Losing a project password leads to the project being unable to open.
- The option **Integrity check** in the **Project settings > Security** dialog box (see Menu Commands, Online Help) saves the project file in a proprietary format. The integrity of the file is verified each time the project is opened.

  NOTE: The project file is not encrypted. To help protect your project from unauthorized access, select the option **Encryption** in the dialog box as indicated above.

Protecting and Saving Applications

Overview

- For protection of boot applications, refer to *Encrypting Boot Applications with Certificates* in the Menu Commands online help.

- For signing a boot application, refer to *Signing a Boot Application* in the Menu Commands online help.

Security for the Runtime System / Controller

What's in This Chapter

Encrypting Communication 167
Handling of Device User Management 167

Encrypting Communication

Overview

Communication with the controller connected in the project should be protected against unauthorized access.

Consideration for HMISCU Controllers

NOTE:

Encrypted communication is not available for HMISCU controllers.

Enforcing Encrypted Communication

For further information, refer to the following sections in the online help:

- Encrypting communication to the controller, page 97
- Changing security settings to add, edit, and remove a user in the online user management of the target device where you are logged in (see Menu Commands, Online Help).
- Managing certificates which should be used for secured TCP communication at application level. For detailed information, refer to the user guide *How To Manage Certificates on the Controller*.
- Managing certificates which are applied by the OPC UA client provided by the controller. For detailed information, refer to the user guide *How to Manage Certificates for OPC UA Client*.
- Managing certificates which are expired, page 249.

NOTE: If the CODESYS Security Agent is installed, refer to the **Software > Programming and Configuring > CODESYS Security Agent** part of the online help.

Handling of Device User Management

Overview

For general information on device user management, refer to the sections *Users and Groups*, page 132 and *Access Rights*, page 137 in this Programming Guide.

User rights management is activated by default in the controllers.

Managing Device User Rights

For further information, refer to the following sections in the online help:

- Editing or Viewing the Users and Groups Management Before any Users and Groups Have Been Established, page 135
- Setting up a New User in the Users and Groups Management of the Controller, page 136
- Loading a Users and Groups Management From a *.dum File, Modifying it, and Later Downloading it to the Controller, page 136
- Modifying Access Rights to Controller Objects in the Users and Groups Management of the Controller, page 138
- Managing device user rights by using the Controller Assistant (see Controller Assistant, User Guide)
- Managing device user rights using the Scripting API, page 874

Resetting Device User Rights

For further information, refer to the following sections in the online help:

- Reset user rights management to default (see Menu Commands, Online Help)
- Reset origin device (see Menu Commands, Online Help)
- Reset via controller Web server (see Modicon M251 Logic Controller, Programming Guide): **MAINTENANCE > USER MANAGEMENT > USER ACCOUNTS MANAGEMENT > RESET TO DEFAULT**
- Reset via Controller Assistant (see Controller Assistant, User Guide)
- Reset without credentials via Controller Assistant (see Controller Assistant, User Guide)

Disabling/Deactivating Device User Rights

For further information, refer to the following sections in the online help:

- Disable user rights management on device (see Menu Commands, Online Help)
- Deactivating with controller web server (see the Programming Guide specific to your controller for compatibility): **MAINTENANCE > USER MANAGEMENT > USER ACCOUNTS MANAGEMENT > DEACTIVATE**
- Deactivating user rights for the simulation device in Logic Builder (see Menu Commands, Online Help)

Including Device User Rights While Cloning the SD Card

The Modicon M241 Logic Controller, Modicon M251 Logic Controller, and the Modicon M262 Logic/Motion Controller provide a clone function that allows you to write the image of the controller to an SD card. Consult the *Programming Guide* specific to your controller, such as the *Cloning a Controller* section for Modicon M262 Logic/Motion Controller (see Modicon M262 Logic/Motion Controller, Programming Guide) or the SD Card section for Modicon M251 Logic Controller (see Modicon M251 Logic Controller, Programming Guide).

This function is also provided by the Controller Assistant (see *Device User Rights Management* in the Controller Assistant User Guide).

Including Device User Rights While Cloning the USB Memory Key

The Modicon M258 Logic Controller, and the Modicon LMC058 Motion Controller provide a clone function that allows you to write the image of the controller to a USB memory key. Consult the *Programming Guide* specific to your controller.

Visualization

What’s in This Chapter

Visualization 170

Visualization

User Management

Visualizations can be protected as described in the **Software > Programming and Configuring > CODESYS Visualization > Setting Up User Management** part of the online help.

Encrypted Communication

For communication between an Internet browser and a controller running a WebVisu, an HTTPS connection with encryption is available which helps to protect the integrity of the displayed data. This function is not available for all controllers. Consult the *Programming Guide* specific to your controller for further information about firewall script commands.

FAQ

What's in This Chapter

Frequently Asked Questions (FAQ) 171

Frequently Asked Questions (FAQ)

Disabling Encryption for Communications

NOTE: It is a good practice to enable encrypted communication, especially in combination with user management enabled, to help protect user credentials.

To disable the encryption of communication to the controller, proceed as follows:

Step	Action
1	If the communication policy for encrypted communication is set to **Enforced encryption**, set it back to **Optional encryption** in the **Change Runtime Security Policy** dialog box, page 115.
2	In the **Communication Settings** tab of the device editor, disable the option **Encrypted communication** in the **Device** menu. As an alternative, you can modify the setting in the **Security Screen** if the CODESYS Security Agent is installed. **Result**: Unencrypted communication is established to the controller. Other clients can also communicate encryption.

New Certificate While the Existing Certificate is Still Valid

It is a good practice to provide a new certificate before the existing certificate expires to help provide seamless encrypted communication. For further information, refer to the chapter *Managing Expired Certificates*, page 249.

Program

What’s in This Part

Program Components 173
Task Configuration 233
Managing Applications 234

Program Components

What's in This Chapter

Program Organization Unit (POU) 173
Function Block 192
Application Objects 204
Application 232

Program Organization Unit (POU)

POU

Overview

The term Program Organizational Unit (POU) refers to the programming objects (programs, function blocks, functions, etc.) used to create a controller application.

As defined in the IEC 61131 standard, a POU is a program, a function block, or a function. However, in this document the term *POU* is used for programming elements in general that can contain IEC code, such as methods, properties, interfaces etc. Whenever this document refers to POUs in the more restricted sense as defined in IEC 61131, the term *POU object* is used.

POU Management

POUs which are managed in the **Global** node of the **Applications tree** are not device-specific but they can be instantiated for use on a device (application). For this purpose, program POUs must be called by a task of the respective application.

But POU also is the name of a certain sub-category of these objects in the **Add Object** menu. Here, it just comprises programs, function blocks, and functions.

Therefore, a POU object in general is a programming unit. It is an object which is managed either non-device-specifically in the **Global** node of the **Applications tree** or directly below an application in the **Applications tree**. It can be viewed and edited in an editor view. A POU object can be a program, function or function block.

It is possible to set certain **Properties** (such as build conditions, etc.) for each particular POU object.

For a description on how to create a POU object, refer to the section *Adding POU Objects to an Application*, page 175.

Besides the POU objects, there are device objects used for running the program on the target system (**Resource**, **Application**, **Task Configuration** etc.). They are managed in the **Applications tree**.

Adding and Calling POU Objects

Introduction

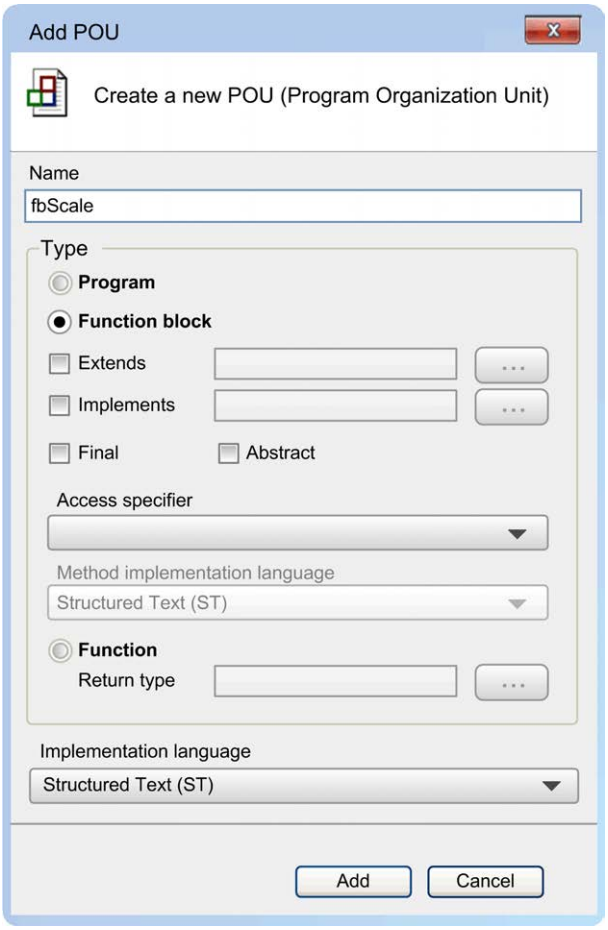
You can add Program Organization Unit (POU) objects to your application in the **Applications tree**.

The different types of POU objects are:

- **Program**: It returns one or several values during operation. All values are retained from the last time the program was run until the next. It can be called by another POU object.
- **Function Block**: It provides one or more values during the processing of a program. As opposed to a function, the values of the output variables and the necessary internal variables shall persist from one execution of the function block to the next. So invocation of a function block with the same arguments (input parameters) need not always yield the same output values.
- **Function**: It yields exactly one data element (which can consist of several elements, such as fields or structures) when it is processed. The call in textual languages can occur as an operator in expressions.

Adding POU Objects to an Application

To add a POU object to the application of the controller, proceed as follows:

Step	Action
1	Right-click the **Application** node of the controller and choose **Add Object > POU**. The two methods are also available in the **Applications tree**. **Result:** The **Add POU** dialog box opens. 
2	In the **Add POU** dialog box, assign a name to your POU object by typing a name in the text field **Name**. **NOTE:** The name must not contain any space characters. If you do not enter a name, a name is given by default. Assigning a meaningful name to a POU may ease the organization of your project.

Step	Action
3	Select the type of POU object you want: • **Program** • **Function Block**: 1. If you want your function block to extend an already existing function block, select **Extends** and click the browser to select the function block you want in the **Input Assistant**. 2. Click the **OK** button. 3. If you want your function block to implement an interface, select **Implements** and click the browser to select the interface you want in the **Input Assistant**. The methods which are defined with the interface are created with the POU. 4. Click the **OK** button. 5. To help prevent deriving access from your function block, activate the option **Final**. The function block cannot be extended by another one. Enables optimized code generation. 6. Activate the option **Abstract** to indicate that the function block has a an incomplete or no implementation and cannot be instantiated. The function block is exclusively used as base function block and the implementation is typically performed in a derived function block. 7. If your function block implements an interface, in the list box **Method implementation language**, select the programming language you want for editing the methods and properties defined in the implemented interface. • **Function**: 1. Click the browse button to select the **Return type** you want in the **Input Assistant**. 2. Click the **OK** button.
4	From the list box **Implementation Language**, select the programming language you want for editing your POU object.
5	Click the **Open** button.

Assigning Programs to a Task

At least one program has to be assigned to a task. To add a program to a task, proceed as follows:

Step	Action
1	Under the node **Task Configuration** of the controller, double-click the task to which you want to add your program. In the **Configuration** tab, click **Add Call**. Alternatively, in the **Applications tree** select the task to which you want to assign your program and click the green plus button. Execute the command **Program Call...** from the list. Click the **...** button. **Result:** The **Input Assistant** dialog box is displayed.
2	In the tab **Categories** of the **Input Assistant** dialog box, select **Programs**.
3	Click to clear the check box **Structured view**.
4	In the **Items** panel, select the POU you want.
5	Click the **OK** button.

Calling POUs

POUs can call other POUs. Recursion however is not allowed (a POU that calls itself).

When a POU assigned to an application calls another POU just by its name (without any namespace, page 740 added), consider the following order of browsing the project for the POU to be called:

1.	current application
2.	**Library Manager** of the current application in the **Tools tree**
3.	**Global** node of the **Applications tree**
4.	**Library Manager** in the **Global** node of the **Tools tree**

If a POU with the name specified in the call is available in a library of the **Library Manager** of the application as well as an object in the **Global** node of the **Applications tree**, there is no syntax for explicitly calling the POU in the **Global** node of the **Applications tree**, just by using its name. In this case move the respective library from the **Library Manager** of the application to the **Library Manager** of the **Global** node of the **Applications tree**. Then you can call the POU from the **Global** node of the **Applications tree** just by its name (and, if needed, that from the library by preceding the library namespace).

Also refer to the chapter *POUs for Implicit Checks*, page 191.

Program

Overview

A program is a POU object which returns one or several values during operation. All values are retained from the last time the program was run until the next. However, there are not separate instances of programs, unlike function blocks. When calling a function block, only the values in the given instance of the function block are modified. The modifications are only affected when the same instance is called again. Program value modifications will be retained until the program gets called again even if called from another POU.

Adding a Program

To add a program to an existing application, select the application node in the **Applications tree**, click the green plus button, and execute the command **POU...**. As an alternative, right-click the **Application** node, and execute the command **Add Object > POU** from the contextual menu. To add an application-independent POU, select the **Global** node of the **Applications tree**, and execute the same commands.

In the **Add POU** dialog box select the **Program** option, enter a name for the program, and select the desired implementation language. Click **Open** to confirm. The editor view for the new program opens and you can start editing the program.

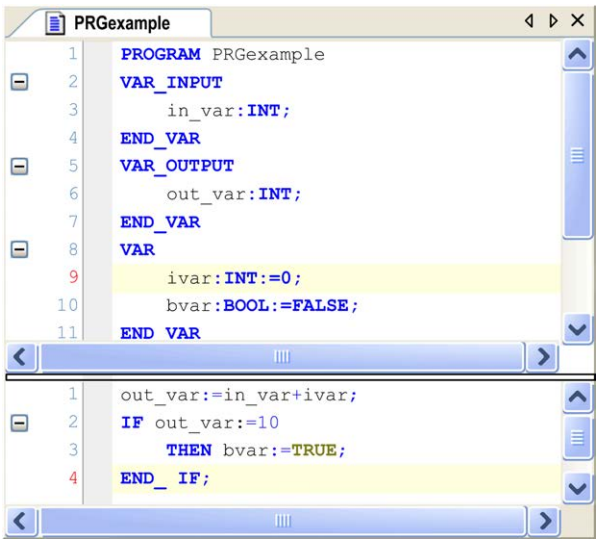
Declaring a Program

Syntax:

PROGRAM <program name>

This is followed by the variable declarations of input, page 567, output, page 567, and program variables. Access variables are available as options as well.

Example of a program



Calling a Program

A program can be called by another POU. However, a program call in a Function, page 179 is not allowed. There are no instances of programs.

If a POU has called a program and if the values of the program have been modified, these modifications will be retained until the program gets called again. This applies even if it is called from within another POU. Consider that this is different from calling a function block. When calling a function block, only the values in the given instance of the function block are modified. The modifications are only affected when the same instance is called again.

In order to set input and/or output parameters in the course of a program call, in text language editors (for example, ST), assign values to the parameters after the program name in parentheses. For input parameters, use `:=` for this assignment, as with the initialization of variables, page 560 at the declaration position. For output parameters, use `=>`. See the following example.

If the program is inserted via the **Input Assistant** using the option **Insert with arguments** in the implementation view of a text language editor, it will be displayed automatically according to this syntax with all parameters, though you do not necessarily have to assign these parameters.

Example for Program Calls

Program in IL:

```
CAL                 PRGexample              (
          in_var:=  33                      )
LD                  PRGexample.out_var
ST                  erg
```

Example with assigning the parameters (**Input Assistant** using the option **Insert with arguments**):

Program in IL with arguments:

```
CAL                 PRGexample              (
          in_var:=  33                      ,
         out_var=>  erg                     )
```

Example in ST

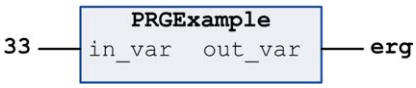
```
PRGexample(in_var:= 33);
erg := PRGexample.out_var;
```

Example with assigning the parameters (**Input Assistant** using the option **Insert with arguments** as described previously):

```
PRGexample (in_var:=33, out_var=>erg );
```

Example in FBD

Program in FBD:



Function

Overview

A function is a POU which yields exactly one data element (which can consist of several elements, such as fields or structures) when it is processed. Its call in textual languages can occur as an operator in expressions.

Adding a Function

To assign the function to an existing application, select the application node in the **Applications tree**, click the green plus button, and execute the command **POU...**. As an alternative, right-click the **Application** node, and execute the command **Add Object > POU** from the contextual menu. To add an application-independent POU, select the **Global** node of the **Applications tree**, and execute the same commands.

In the **Add POU** dialog box, select the **Function** option. Enter a **Name** (<function name>) and a **Return Data Type** (<data type>) for the new function and select the desired implementation language. To choose the return data type, click the button **...** to open the **Input Assistant** dialog box. Click **Open** to confirm. The editor view for the new function opens and you can start editing.

Declaring a Function

Syntax:

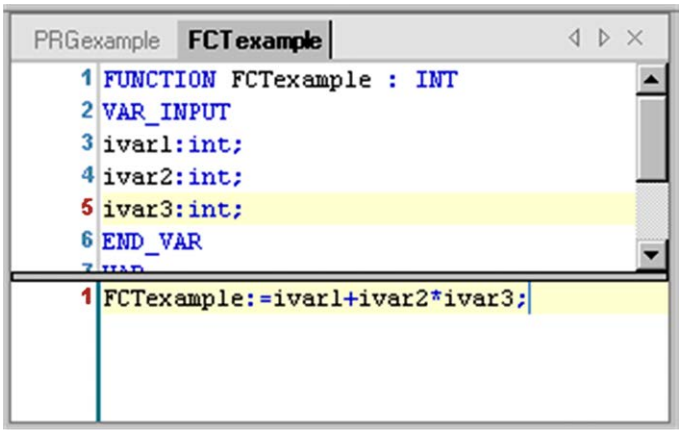
FUNCTION <function name> : <data type>

This is followed by the variable declarations of input and function variables.

Assign a result to the function. Therefore, the function name is used as an output variable.

Do not declare local variables as `RETAIN` or `PERSISTENT` in a function because this will have no effect.

Example of a function in ST: this function takes 3 input variables and returns the product of the last 2 added to the first one.



Calling a Function

The call of a function in ST can appear as an operand in expressions.

In IL, you can position a function call only within actions of a step or within a transition.

Functions (in contrast to a program or function block) contain no internal state information, that is, invocation of a function with the same arguments (input parameters) always will yield the same values (output). For this reason, functions may not contain global variables and addresses.

Example of Function Calls in IL

Function calls in IL;

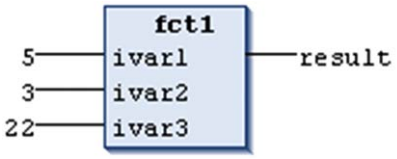
```
LD              5
Fct             3               ,
                22
ST              result
```

Example of Function Calls in ST

```
result := fct1(5,3,22);
```

Example of Function Calls in FBD

Function calls in FBD:



Example:

```
fun(formal1 := actual1, actual2); // -> error message
fun(formal2 := actual2, formal1 := actual1); // same
semantics as the following:
fun(formal1 := actual1, formal2 := actual2);
```

According to the IEC 61131-3 standard, functions can have additional outputs. They can be assigned in the call of the function. In ST, for example, according to the following syntax:

out1 => <output variable 1> | out2 => <output variable 2> | ...further output variables

Example

Function `fun` is defined with 2 input variables `in1` and `in2` and two output variables *out1* and *out2*. The output values of `fun` are written to the locally declared variables `loc1` and `loc2`.

```
fun(in1 := 1, in2 := 2, out1 => loc1, out2 => loc2);
```

Method

Overview

A method is a language element similar to a function that can be used in a context of a function block, page 192. It can be regarded as a function which contains an instance of the respective function block. Such as a function, a method has a return value, and its own declaration part for temporary variables and parameters.

Also as a means of object-oriented programming, you can use interfaces, page 186 to organize the methods available in a project.

NOTE: When copying or moving a method or property from a POU to an interface, the contained implementations are deleted automatically. When copying or moving from an interface to a POU, you are requested to specify the desired implementation language.

Inserting a Method

To assign a method to a function block or interface, select the corresponding function block or interface node in the **Applications tree**, click the green plus button and execute the command **Method**. Alternatively, you can right-click the function block or interface node and execute the command **Add Object > Method** from the contextual menu.

In the **Add Method** dialog box, enter a **Name**, the desired **Return Type**, the **Implementation Language**, and the **Access Specifier** (see below). If the method does not have an implementation and the implementation is provided by the derived function block, select the option **Abstract**. For choosing the return data type, click the button **...** to open the **Input Assistant...** dialog box.

Access specifier: For compatibility reasons, access specifiers are optional. The specifier **PUBLIC** is available as an equivalent for having set no specifier.

Alternatively, choose one of the options from the selection list:

- **PRIVATE**: The access on the method is restricted to the function block.
- **PROTECTED**: The access on the method is restricted to the function block and its derivation.
- **INTERNAL**: The access on the method is restricted to the present namespace (the library).

Click **Open** to confirm. The method editor view opens.

Input Assistance when Creating Inheriting Function Blocks

Object-oriented programming is facilitated using inheritance within function blocks: When you execute **Add Object** on a function block that inherits from another function block, the **Action**, **Method**, **Property**, and **Transition** elements used in the base function block are listed for selection:

- **Action**, **Method**, **Property**, and **Transition** elements with **Access specifier** = **PUBLIC**, **PROTECTED**, and **INTERNAL** defined in the base function block are available for selection. You can adapt the definition for the inherited object. In the inherited object, the same **Access specifier** is assigned as to the source elements.
- **Action**, **Method**, **Property**, and **Transition** elements with **Access specifier** = **PRIVATE** are not available for selection because access is restricted to the base function block.

Declaring a Method

Syntax:

METHOD <access specifier> <method name> : <return data type>VAR_INPUT ... END_VAR

For a description on how to declare interface handling methods, refer to the *Interface* chapter, page 186.

Calling a Method

Method calls are also named virtual function calls. For further information, refer to the chapter *Method Invocation*, page 200.

Note the following for calling a method:

- The data of a method is temporary and only valid during the execution of the method (stack variables). Therefore, the variables and function blocks declared in a method are reinitialized at each call of the method.
- Methods defined in an interface, page 186 are only allowed to have input, output, and input/output variables, but no body (implementation part).
- Methods such as functions can have additional outputs. They must be assigned during *method invocation*, page 200.
- The declared access specifier defines how the method can be called:
  - **INTERNAL**: The method can be called within its own namespace.
  - **PROTECTED**: The method can be called within its own POU and its derivatives.
  - **PRIVATE**: The method can be called within its own POU.
  - **PUBLIC**: No restrictions apply for calling the method.

  NOTE: If no return value is defined for the method, the value of the first output is returned. If no output is available for the method, a compiler error message is generated.

Implementing a Method

Note the following for implementing a method:

- In the body of a method, access to the function block instance variables is allowed.
- If necessary, use the THIS pointer, page 203 which always points on the present instance.
- `VAR_TEMP` variables of the function block cannot be accessed in a method.
- A method can call itself recursively.

Calling a Method

Use the following syntax for calling a method:

```
<return value variable> := <POU name> . <method name> (
<method input name> := <variable name> (, <further method
input name> := <variable name> )* );
```

Declaration example:

```
METHOD PUBLIC DoIt : BOOL
VAR_INPUT
    iInput_1 : DWORD;
    iInput_2 : DWORD;
    sInput_3 : STRING(12);
END_VAR
```

Call example:

```
bFinishedMethod := fbInstance.DoIt(sInput_3 :='Hello World
', iInput_2 := 16#FFFF,iInput_1 := 16);
```

NOTE: When the method is called, the return value of the method is assigned, for example, to variables declared locally. When you omit the names of the input variables, make sure to obey the declaration order.

Declaration example:

```
METHOD PUBLIC DoIt : BOOL
VAR_INPUT
    iInput_1 : DWORD;
    iInput_2 : DWORD;
    sInput_3 : STRING(12);
END_VAR
```

Call example:

```
bFinishedMethod := fbInstance.DoIt( 16, 16#FFFF,'Hello World
');
```

Recursive Method Call

Within the implementation, a method can call itself, either directly by using the `THIS` pointer, or by using a local variable of the method that temporarily instantiates the relevant function block.

A recursive call results in a compiler message. The compiler message will not be issued if the method is provided with the pragma `{attribute 'estimated-stackusage:'<estimated_stack_size_in_bytes>'}`. Refer to the `Attribute estimated-stack-usage` chapter, page 606 for an implementation example.

Thus, specifying the method name is insufficient for recursive method calls. In this case, the following compiler message is issued:

```
Program name, function or function block instance expected
instead of
```

NOTE: Recursive calls are inherently difficult to implement and debug, and may lead to unwanted side effects such as memory issues and watchdog timeouts.

NOTICE
UNINTENDED APPLICATION ISSUES • Only implement recursive algorithms where necessary. • Be sure to have a solid understanding of recursive implementation techniques, and document well the supporting code. **Failure to follow these instructions can result in equipment damage.**

Special Methods for a Function Block

Method	Description
`Init`	A method named `FB_init` is by default declared implicitly, but can also be declared explicitly. It contains initialization code for the function block as declared in the declaration part of the function block. Refer to `FB_init` method, page 579.
`Reinit`	If a method named `FB_reinit` is declared for a function block instance, it is called after the instance has been copied (like during **Online Change**) and will reinitialize the new instance module. Refer to `FB_init`, `FB_reinit` methods, page 579.
`Exit`	If an exit method named `FB_exit` is desired, it has to be declared explicitly. There is no implicit declaration. The `Exit` method is called for each instance of the function block before a new download, a reset or during online change for all moved or deleted instances. Refer to `FB_exit` method, page 579.

Properties, page 184 and interface properties, page 187 each consist of a `Set` and/or a `Get` accessor method.

Method Call Also When Application Is Stopped

In the device description file, it can be defined that a certain method should always be called task-cyclically by a certain function block instance (of a library module). If this method has the following input parameters, it is processed also when the active application is not running.

Example

```
VAR_INPUT
pTaskInfo : POINTER TO DWORD;
pApplicationInfo: POINTER TO _IMPLICIT_APPLICATION_INFO;
END_VAR
```

The programmer can check the application status via `pApplicationInfo`, and can define what should happen.

```
IF pApplicationInfo^.state = RUNNING THEN <instructions>
END_IF
```

Property

Overview

A property in extension to the IEC 61131-3 standard is available as a means of object-oriented programming. It consists of a pair of accessor methods (`Get`, `Set`). They allow encapsulating a read or write access to variables declared inside of a POU or a GVL into a function call, while keeping the syntax of a variable access.

To insert a property as an object below a program, page 177, a function block, page 192, a GVL, page 205, or an interface, page 186 node, select the node in the **Applications tree**, click the green plus button, and execute the command **Property**. As an alternative, right-click the node and execute the command **Add Object > Property** from the contextual menu.

In the **Add Property** dialog box specify the **Name**, **Return Type**, desired **Implementation Language**, and optionally an **Access Specifier**.

The following access specifiers are available as for methods, page 181:

- **PUBLIC**
- **PRIVATE**
- **PROTECTED**
- **INTERNAL**

- **ABSTRACT**

  NOTE: Properties can also be declared within interfaces.

Object-oriented programming is facilitated using inheritance within function blocks: When you execute **Add Object** on a function block that inherits from another function block, the **Action**, **Method**, **Property**, and **Transition** elements used in the base function block are listed for selection:

- **Action**, **Method**, **Property**, and **Transition** elements with **Access specifier** = **PUBLIC**, **PROTECTED**, and **INTERNAL** defined in the base function block are available for selection. You can adapt the definition for the inherited object. In the inherited object, the same **Access specifier** is assigned as to the source elements.
- **Action**, **Method**, **Property**, and **Transition** elements with **Access specifier** = **PRIVATE** are not available for selection because access is restricted to the base function block.

`Get` and `Set` Accessors of a Property

Two special methods, page 181, named accessor, are inserted automatically in the **Applications tree** below the property object. You can delete one of them if the property should only be used for writing or only for reading. An accessor, like a property (see previous paragraph), can get assigned an access modifier in the declaration part, or via the **Add Object** dialog box, when explicitly adding the accessor.

- The `Set` accessor is called when the property is written.
- The `Get` accessor is called when the property is read.

Example:

Function block `FB1` has a property `seconds` that uses a local variable `milli`. This variable is determined by the properties `Get` and `Set`:

`Get` implementation example
```
seconds := milli / 1000;
```

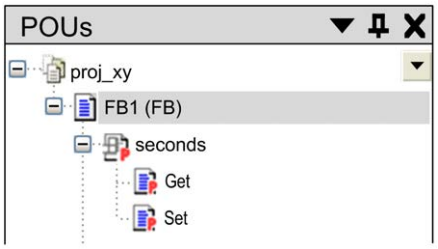
`Set` implementation example
```
milli := seconds * 1000;
```

You can write the property of the function block (`Set` method), for example by `fbinst.seconds := 22;`.

(`fbinst` is the instance of `FB1`).

You can read the property of the function block (`Get` method) for example by `testvar := fbinst.seconds;`.

In the following example, property `seconds` is assigned to function block `FB1`:



A property can have additional local variables but no additional inputs and - in contrast to a function, page 179 or method, page 181 - no additional outputs.

NOTE: When copying or moving a method or property from a POU to an interface, the contained implementations are deleted automatically. When copying or moving from an interface to a POU, you are requested to specify the desired implementation language.

Monitoring a Property

A property can be monitored in online mode either with help of inline monitoring, page 357 or with help of a watch list, page 423. The precondition for monitoring a property is the addition of the pragma `{attribute 'monitoring' := 'variable'}` (refer to the chapter Attribute Monitoring, page 615).

Interface

Overview

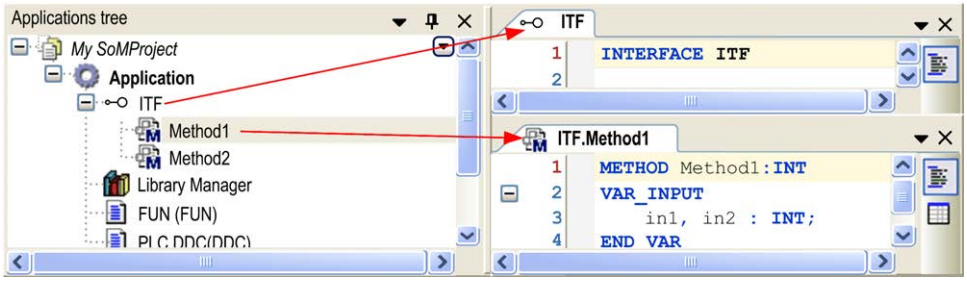
The use of interfaces is a means of object-oriented programming. An interface POU defines a set of methods, page 181 and properties, page 184 without an implementation. An interface can be described as an empty shell of a function block, page 192. It must be implemented, page 198 in the declaration of the function block in order to get realized in the function block instances. A function block can implement one or several interfaces.

The same method can be realized with identical parameters but different implementation code by different function blocks. Therefore, an interface can be used/called in any POU without the need that the POU identifies the particular function block that is concerned.

Example of Interface Definition and Usage in a Function Block

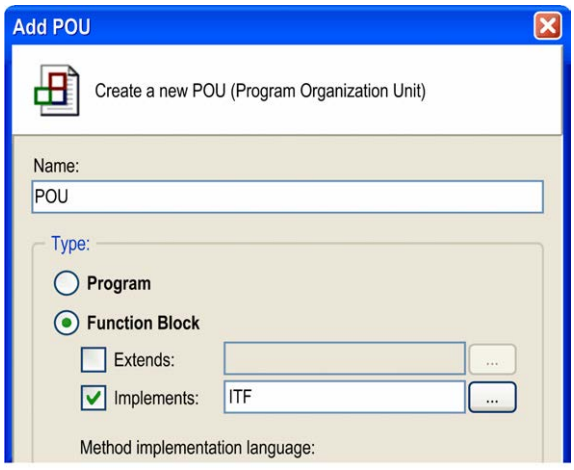
An interface `IFT` is inserted below an application. It contains 2 methods `Method1` and `Method2`. Neither the interface nor the methods contain any implementation code. Just the declaration part of the methods is to be filled with the desired variable declarations:

Interface with 2 methods:



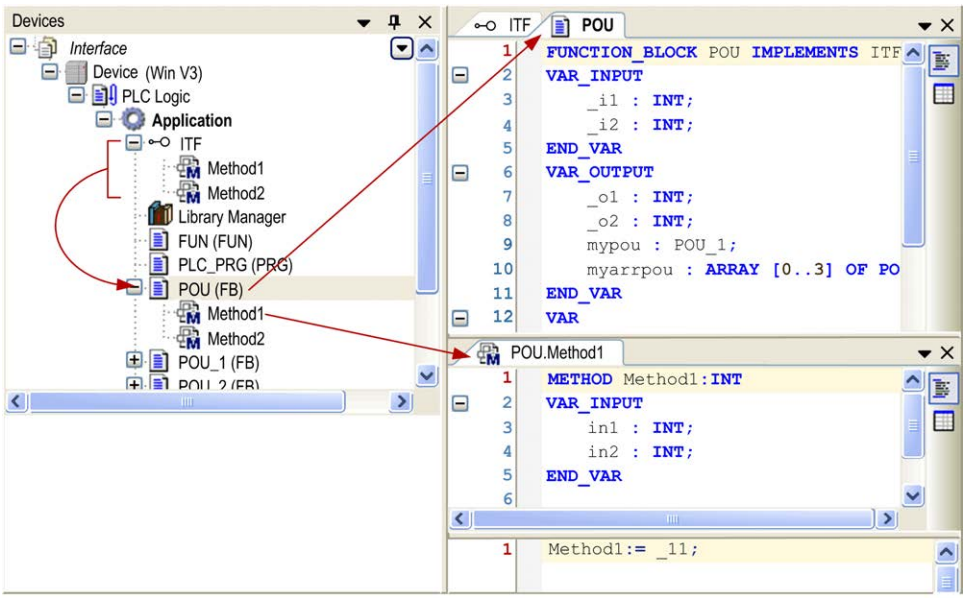
1 or multiple function blocks can now be inserted, implementing the above defined interface `ITF`.

Creating a function block implementing an interface



When function block `POU` is added to the **Applications tree**, the methods `Method1` and `Method2` are automatically inserted below as defined by `ITF`. Here they can be filled with function block-specific implementation code.

Using the interface in the function block definition



An interface can extend other interfaces by using `EXTENDS` (see following example *Example for Extending an Interface*, page 188) in the interface definition.

Interface Properties

An interface can also define an interface property, consisting of the accessor methods `Get` and/or `Set`. For further information on properties, refer to the chapter *Property*, page 184. A property in an interface like the possibly included methods is just a prototype that means it contains no implementation code. Like the methods, it is automatically added to the function block, which implements the interface. There it can be filled with specific programming code.

Considerations

Consider the following:

- It is not allowed to declare variables within an interface. An interface has no body (implementation part) and no actions. Just a collection of methods is defined within an interface and those methods are only allowed to have input variables, output variables, and input/output variables.
- Variables declared with the type of an interface are treated as references.
- A function block implementing an interface must have assigned methods and properties which are named exactly as they are in the interface. They must contain identically named inputs, outputs, and inputs/outputs.

  NOTE: When copying or moving a method or property from a POU to an interface, the contained implementations are deleted automatically. When copying or moving from an interface to a POU, you are requested to specify the desired implementation language.

Inserting an Interface

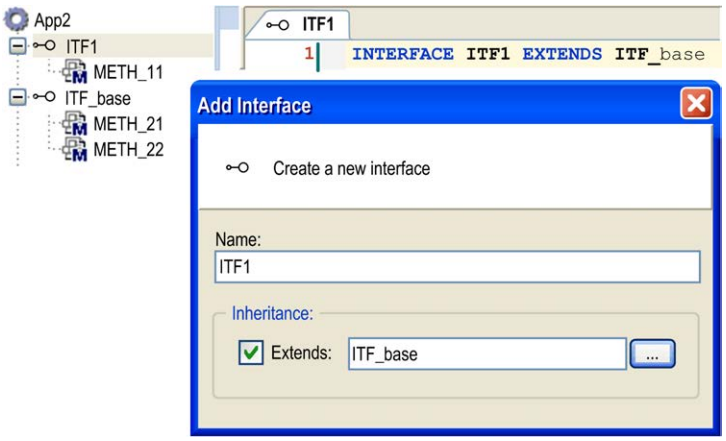
To add an interface to an application, select the **Application** node in the **Applications tree**, click the green plus button and select **Add Other Objects... > Interface**. Alternatively, execute the command **Add Object > Interface**. If you select the node **Global** before you execute the command, the new interface is available for all applications.

In the **Add Interface** dialog box, enter a name for the new interface (<interface name>). Optionally you can activate the option **Extends:** if you want the current interface to be an extension, page 197 of another interface.

Example for Extending an Interface

If `ITF1` extends `ITF_base`, all methods described by `ITF_base` will be automatically available in `ITF1`.

Extending an interface



Click **Add** to confirm the settings. The editor view for the new interface opens.

Declaring an Interface

Syntax

INTERFACE <interface name>

For an interface extending another one:

INTERFACE <interface name> EXTENDS <base interface name>

Example

```
INTERFACE interface1 EXTENDS interface_base
```

Adding the Desired Collection of Methods

To complete the definition of the interface, add the desired collection of methods. For this purpose, select the interface node in the **Applications tree** and execute the command **Interface method...**. The **Add Interface Method** dialog box opens for defining a method to be part of the interface. Alternatively, select the interface node in the **Applications tree**, click the green plus button and select **Interface Method**. Add as many methods as desired and remember that these methods are only allowed to have input variables, output variables, and input/output variables, but no body (implementation part).

Action

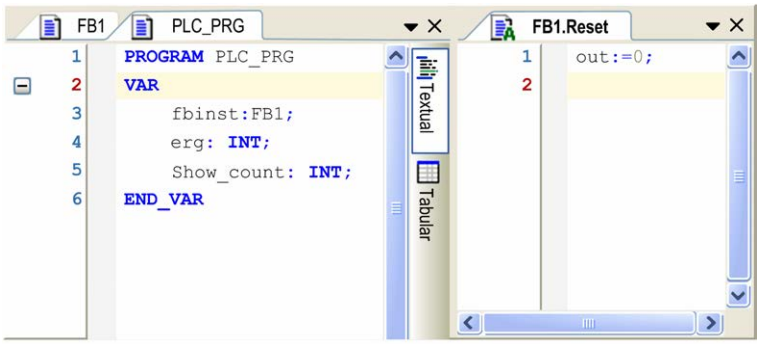
Overview

You can define actions and assign them to function blocks, page 192 and programs, page 177. An action is an additional implementation. It can be created in a different language than the basic implementation. Each action is given a name.

An action works with the data of the function block or program to which it belongs. It uses the input/output variables and local variables defined and does not contain its own declarations.

Example of an Action of a Function Block

The following illustration shows an action in FB



In this example, each call of the function block `FB1` increases or decreases the output variable `out`, depending on the value of the input variable `in`. Calling action `Reset` of the function block sets the output variable `out` to 0. The same variable `out` is written in both cases.

Inserting an Action

To add an action, select the respective program or function block node in the **Applications Tree** or in the **Global** node of the **Applications Tree**, click the green plus button, and execute the command **Action...**. Alternatively, right-click the program or function block node, and execute the command **Add Object > Action**. In the **Add Action** dialog box, define the action **Name** and the desired **Implementation Language**.

Object-oriented programming is facilitated using inheritance within function blocks: When you execute **Add Object** on a function block that inherits from another function block, the **Action**, **Method**, **Property**, and **Transition** elements used in the base function block are listed for selection:

- **Action**, **Method**, **Property**, and **Transition** elements with **Access specifier** = **PUBLIC**, **PROTECTED**, and **INTERNAL** defined in the base function block are available for selection. You can adapt the definition for the inherited object. In the inherited object, the same **Access specifier** is assigned as to the source elements.
- **Action**, **Method**, **Property**, and **Transition** elements with **Access specifier** = **PRIVATE** are not available for selection because access is restricted to the base function block.

Calling an Action

Syntax
```
<Program_name>.<Action_name>
```

or
```
<Instance_name>.<Action_name>
```

Consider the notation in FBD (see the following example).

If it is required to call the action within its own block, that is the program or function block it belongs to it is sufficient to use the action name.

Examples

This section provides examples for the call of the above described action from another POU.

Declaration for all examples:

```
PROGRAM PLC_PRG
VAR
    Inst : Counter;
END_VAR
```

Call of action `Reset` in another POU, which is programmed in IL:

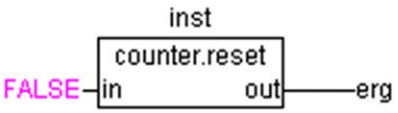
```
CAL Inst.Reset(In := FALSE)
LD Inst.out
ST ERG
```

Call of action `Reset` in another POU, which is programmed in ST:

```
Inst.Reset(In := FALSE);
Erg := Inst.out;
```

Call of action `Reset` in another POU, which is programmed in FBD:

Action in FBD



NOTE: The IEC standard does not recognize actions other than actions of the sequential function chart (SFC). These actions are an essential part containing the instructions to be processed at the particular steps of the chart.

Transition

Overview

You can use transition objects as transition elements in a program or function block implemented in SFC.

For further information, refer to the description of steps and transitions in the chapter *SFC Elements / ToolBox*, page 337.

Object-oriented programming is facilitated using inheritance within function blocks: When you execute **Add Object** on a function block that inherits from another function block, the **Action**, **Method**, **Property**, and **Transition** elements used in the base function block are listed for selection:

- **Action**, **Method**, **Property**, and **Transition** elements with **Access specifier** = **PUBLIC**, **PROTECTED**, and **INTERNAL** defined in the base function block are available for selection. You can adapt the definition for the inherited object. In the inherited object, the same **Access specifier** is assigned as to the source elements.
- **Action**, **Method**, **Property**, and **Transition** elements with **Access specifier** = **PRIVATE** are not available for selection because access is restricted to the base function block.

POUs for Implicit Checks

Overview

You have to add special POUs below an application to provide implicitly available check functions. They verify array and range boundaries, the validity of pointers and verify the presence of division by zero during runtime.

NOTE: The availability of the check functions have a runtime performance impact.

NOTE: For the check functions to verify the code of uncompiled referenced libraries, you have to enter the **Compiler defines** string `checks_in_libs` in the **Build** tab of the **Properties** dialog box (see Menu Commands, Online Help) of the application.

To add monitoring functions under an application, select the application node in the **Applications tree** and click the green plus button or execute the command **Add Object > POU for implicit checks...**.

The dialog box allows you to activate the following check functions for the selected application:

Category	Added functions
Bound Checks	CheckBounds (for arrays), page 651
Division Checks	• CheckDivDInt, page 683 • CheckDivLInt, page 683 • CheckDivReal, page 683 • CheckDivLreal, page 683
Range Checks	• CheckRangeSigned, page 666 • CheckRangeUnsigned, page 666
LRange Checks	• CheckLRangeSigned, page 666 • CheckLRangeUnsigned, page 666
Pointer Checks	CheckPointer, page 649

When you insert a check POU, the option is no longer available in the dialog box to help avoid a double insertion. If the different types of check POUs have been added below the application, the **Add Object** dialog box does not provide the **POUs for implicit checks** option any longer.

⚠CAUTION

INCORRECT FUNCTIONALITY

Do not modify the declaration part of an implicit check function in order to maintain its functional integrity.

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

NOTE: You may, however, add local variables to the declaration part of an implicit check function.

NOTE: After having removed implicit check function (such as CheckBounds) from your application, no **Online Change** is possible, just a download. A corresponding message is displayed.

Function Block

General Information

Overview

A function block is a POU, page 173 which provides 1 or more values during the processing of a controller program. As opposed to a function, the values of the output variables and the necessary internal variables shall persist from one execution of the function block to the next. Therefore, invocation of a function block with the same arguments (input parameters) need not always yield the same output values.

In addition to the functionality described by standard IEC11631-3, object-oriented programming is supported and function blocks can be defined as extensions, page 197 of other function blocks. They can include interface, page 198 definitions concerning Method invocation, page 200. Therefore, inheritance can be used when programming with function blocks.

A function block always is called via an instance, page 193, which is a reproduction (copy) of the function block.

Adding a Function Block

To add a function block to an existing application, select the respective node in the **Applications tree**, click the green plus button and select **POU...**. Alternatively you can right-click the node and execute the command **Add Object > POU**. To create a function block that is independent of an application, select the **Global** node of the **Applications tree**.

In the **Add Object** dialog box, select the option **Function Block**, enter a function block **Name** (<identifier>) and choose the desired **Implementation Language**.

Additionally, you can set the following options:

Option	Description
Extends	Enter the name of another function block available in the project, which should be the base for the current one. For details, refer to *Extension of a Function Block*, page 197.
Implements	Enter the names of interfaces, page 186 available in the project, which should be implemented in the current function block. You can enter several interfaces separated by commas. For details, refer to *Implementing Interfaces*, page 198.
Access specifier	For compatibility reasons, access specifiers are optional. Specifier **PUBLIC** is available as an equivalent for having set no specifier. Alternatively, choose one of the options from the selection list: • **INTERNAL**: The access on the function block is restricted to the current namespace (the library). • **FINAL**: Deriving access is not possible that is the function block cannot be extended by another one. Enables optimized code generation. **NOTE:** The access specifiers are valid as of compiler version 3.4.4.0 and thus can be used as identifiers in earlier versions. For further information, refer to the EcoStruxure Machine Expert/ CODESYS compiler version mapping table in the corresponding EcoStruxure Machine Expert Compatibility and Migration User Guide.
Method implementation language	Choose the desired programming language for all method and property objects created via the interface implementation, independently from that set for the function block itself.

Click **Add** to confirm the settings. The editor view for the new function block opens and you can start editing.

Declaring a Function Block

Syntax

FUNCTION_BLOCK <access specifier> <function block name> | EXTENDS <function block name> | IMPLEMENTS <comma-separated list of interface names>

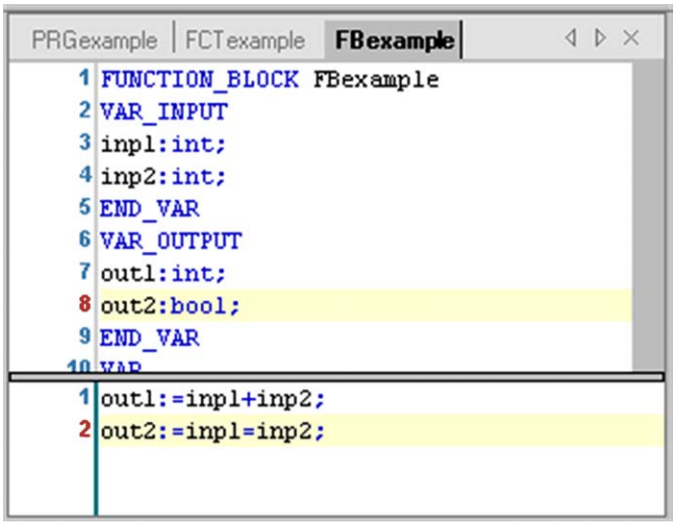
This is followed by the declaration of the variables. You can also group the inputs and outputs for quick fading out and in when the function block is used in an FBD or LD editor. Also refer to the chapter *Attribute Pingroup*, page 625.

Example

`FBexample` shown in the following figure has 2 input variables and 2 output variables `out1` and `out2`.

`out1` is the sum of the 2 inputs, `out2` is the result of a comparison for equality.

Example of a function block in ST



Function Block Instance

Overview

Function blocks are called, page 194 through an instance which is a reproduction (copy) of a function block, page 192.

Each instance has its own identifier (instance name), and a data structure containing its inputs, outputs, and internal variables.

Instances like variables are declared locally or globally. The name of the function block is indicated as the data type of an identifier.

Syntax for Declaring a Function Block Instance

<identifier>:<function block name>;

Example

Declaration (for example, in the declaration part of a program) of instance `INSTANCE` of function block `FUB`:

```
INSTANCE: FUB;
```

The declaration parts of function blocks and programs can contain instance declarations.

Calling a Function Block

Overview

Function blocks, page 192 are called through a function block instance. Thus a function block instance has to be declared locally or globally. Refer to the chapter *Function Block Instance*, page 193 for information on how to declare.

Then the desired function block variable can be accessed using the following syntax.

Syntax

<instance name>.<variable name>

Considerations

- Only the input and output variables of a function block can be accessed from outside of a function block instance, not its internal variables.
- Access to a function block instance is limited to the POU, page 173 in which it was declared unless it was declared globally.
- At calling the instance, the desired values can be assigned to the function block parameters. See the following paragraph *Assigning Parameters at Call*.
- The input / output variables (`VAR_IN_OUT`) of a function block are passed as pointers.
- In SFC, function block calls can only take place in steps.
- The instance name of a function block instance can be used as an input parameter for a function or another function block.
- All values of a function block are retained until the next processing of the function block. Therefore, function block calls do not always return the same output values, even if done with identical arguments.

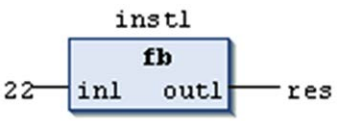
  NOTE: If at least 1 of the function block variables is a remanent variable, the total instance is stored in the retain data area.

Examples for Accessing Function Block Variables

Assume: Function block `fb` has an input variable `in1` of the type INT. See here the call of this variable from within program `prog`. See declaration and implementation in ST:

```
PROGRAM prog
VAR
inst1:fb;
END_VAR
inst1.in1:=22;   (* fb is called and input variable in1 gets
assigned value 22 *)
inst1(); (* fb is called, this is needed for the following
access on the output variable *)
res:=inst1.out1; (* output variable of fb is read *)
```

Example of a function block call in FBD:



Assigning Parameters at Call

In the text languages IL and ST, you can set input and/or output parameters immediately when calling the function block. The values can be assigned to the parameters in parentheses after the instance name of the function block. For input parameters, this assignment takes place using `:=` as with the initialization of variables, page 560 at the declaration position. For output parameters, `=>` is to be used.

Example of a Call with Assignments

In this example, a timer function block (instance `CMD_TMR`) is called with assignments for the parameters `IN` and `PT`. Then the result variable `Q` is assigned to the variable `A`. The result variable is addressed with the name of the function block instance, a following point, and the name of the variable:

```
CMD_TMR(IN := %IX5, PT := T#100MS);
A:=CMD_TMR.Q;
```

Example of Inserting Via Input Assistant with Arguments

If the instance is inserted via **Input Assistant** with the option **With arguments** in the implementation view of an ST or IL POU, it is displayed automatically according to the syntax showed in the following example with all of its parameters, though it is not necessarily required to assign these parameters.

For the previously mentioned example, the call would be displayed as follows.

```
CMD_TMR(in:=, pt:=, q=>)
-> fill in, e.g.:
CMD_TMR(in:=bvar, pt:=t#200ms, q=>bres);
```

Configuring a Memory Reserve for the Online Change of Function Blocks

Overview

You can configure a memory reserve for the online change of function blocks. After you have made modifications on the declaration of a function block (in particular after you have added a new variable), and you perform an online change, it is no longer necessary to copy the instances of the function block to a new memory area. Thus, the online change is performed faster and fewer errors are detected.

If the memory reserve you configured is depleted, a message is displayed before the online change is performed.

It is a good practice to configure the memory reserve for a function block before you download the application to the controller. If you configure the memory reserve when the application is already located on the controller, then you have to perform an online change that is even more time-consuming.

Procedure of Configuring a Memory Reserve for the Online Change of Function Blocks

If you intend to perform later modifications on a function block that would require the function block instance to be copied to other memory locations when executing an online change, then you can use the **Online Change Memory Reserve Settings** command.

By default, the command is not available in the menus. Add this command with the **Tools > Customize** menu (see Menu Commands, Online Help).

Step	Action	Comment
1	Execute the command **View > Online Change Memory Reserve Settings**	**Result**: The **Online Change Memory Reserve** view opens.
2	From the list of applications, select the suitable application.	–
3	Execute the **Generate Code** command from the **Build** menu.	–
4	Click the button **Scan Application**.	–
5	From the **Function Blocks** area, select the entry **All**.	**Result**: The function blocks of the application are displayed in the middle of the **Online Change Memory Reserve** view.
6	Select the function block for which you want to configure a memory reserve.	If the application has not been downloaded to the controller yet, then the input field **Memory reserve (in bytes)** is editable. If the application is already available on the controller, then click the **Edit** button in the **Enable editing** area. **NOTE:** If you modify the memory reserve of an application that is already available on the controller, then the instances of all affected function blocks must be copied.
7	Enter a value for the **Memory reserve (in bytes)**, and click the **Apply for selection** button.	**Result**: The value you entered is displayed in the **Memory Reserve** cell of the table.
8	Execute the **Generate Code** command from the **Build** menu.	–
9	Click the button **Scan Application**.	**Result**: The values of the cells **Size**, **Instance Count**, **Additional memory for all instances** and **Remaining size of memory reserve Memory Reserve** are updated in the list for the configured function block.

When you download the application to the controller, the memory size that is reserved for the function block is calculated by the present size of the function block plus the memory reserve you configured in the **Online Change Memory Reserve** view.

Future modifications to the function block can be downloaded to the controller with the online change without having to copy all instances of the function block to a new memory area.

Extension of a Function Block

Overview

Supporting object-orientated programming, a function block can be derived from another function block. This means a function block can extend another, thus automatically getting the methods/properties/actions/transitions and variables of the basing function block in addition to its own.

The extension is performed by using the keyword `EXTENDS` in the declaration of a function block. You can choose the `EXTENDS` option already during adding a function block to the project via the **Add Object** dialog box.

For further information, refer to the sections describing how to add a method, page 181, a property, page 184, an action, page 189, or a transition, page 190.

Syntax

FUNCTION_BLOCK <function block name> EXTENDS <function block name>

This is followed by the declaration of the variables.

Example

Definition of function block `fbA`

```
FUNCTION_BLOCK fbA
VAR_INPUT
    x:int;
END_VAR
...
```

Definition of function block `fbB`

```
FUNCTION_BLOCK fbB EXTENDS fbA
VAR_INPUT
    ivar: INT := 0;
END_VAR
...
```

Extension by `EXTENDS`

Extension by `EXTENDS` means:

- `fbB` contains all data and methods which are defined by `fbA`. An instance of `fbB` can now be used in any context where a function block of type `fbA` is expected.
- `fbB` is allowed to override the methods/properties/actions/transitions defined in `fbA`. This means: `fbB` can declare a method with the same name and the same inputs and output as declared by `A`.
- `fbB` is not allowed to declare function block variables with the same name as used in `fbA`. In this case, the compiler will generate an error message.
- `fbA` variables and methods can be accessed directly within an `fbB` scope by using the SUPER pointer, page 201 (`SUPER^.<method>`).

  NOTE: A function block can implement multiple interfaces, but it can only extend a single function block.

Example

```
FUNCTION_BLOCK FB_Base
VAR_INPUT
END_VAR
```

```
VAR_OUTPUT
    iCnt : INT;
END_VAR
VAR
END_VAR
THIS^.METH_DoIt();
THIS^.METH_DoAlso();

    METHOD METH_DoIt : BOOL
    VAR
    END_VAR
    iCnt := -1;
    METH_DoIt := TRUE;

    METHOD METH_DoAlso : BOOL
    VAR
    END_VAR
    METH_DoAlso := TRUE;
FUNCTION_BLOCK FB_1 EXTENDS FB_Base
VAR_INPUT
END_VAR
VAR_OUTPUT
END_VAR
VAR
END_VAR
// Calls the method defined under FB_1
THIS^.METH_DoIt();
THIS^.METH_DoAlso();
// Calls the method defined under FB_Base
SUPER^.METH_DoIt();
SUPER^.METH_DoAlso();
    METHOD METH_DoIt : BOOL
    VAR
    END_VAR
    iCnt := 1111;
    METH_DoIt := TRUE;
PROGRAM PLC_PRG
VAR
    Myfb_1: FB_1;
    iFB: INT;
    iBase: INT;
END_VAR
Myfb_1();
iBase := Myfb_1.iCnt_Base;
iFB := Myfb_1.iCnt_THIS;
```

Implementing Interfaces

Overview

In order to support object-oriented programming, a function block can implement several interfaces, page 186 which allows you to use methods, page 181.

Syntax

FUNCTION_BLOCK <function block name> IMPLEMENTS <interface_1 name>|, <interface_2 name>, ..., <interface_n name>

A function block that implements an interface must contain all methods and properties (interface properties, page 187) defined by this interface. This includes name, inputs, and the output of the particular method or property which must be exactly the same.

For this purpose - when creating a new function block implementing an interface - automatically all methods and properties defined in this interface will be inserted below the new function block in the **Applications Tree**.

> **NOTE:** If afterwards, methods are added to the interface definition, they will not be added automatically in the concerned function blocks and thus produce compiler errors. Execute the command Implement interfaces... (see Menu Commands, Online Help) to perform this update explicitly.
>
> When you use inheritance for function blocks, consider that if methods or attributes were created by inheritance of an interface, either implement them, or delete them if the implementation of the base function block is to be used. Pragma attributes are automatically inserted. They are detected during compilation and messages are generated to remind you that the inherited methods or properties have to be verified. Delete the pragma attributes after you have completely implemented the new function block.

For further information, refer to the description of the **Implement interfaces...** command (see Menu Commands, Online Help).

Example

`INTERFACE I1` includes method `GetName`:
```
METHOD GetName : STRING
```

Function blocks `A` and `B` each implement interface `I1`:
```
FUNCTION_BLOCK A IMPLEMENTS I1
FUNCTION_BLOCK B IMPLEMENTS I1
```

Thus in both function blocks the method `GetName` has to be available and will be inserted automatically below each when the function blocks are inserted in the **Applications Tree**.

Consider a declaration of a variable of type `I1`:
```
FUNCTION DeliverName : STRING
VAR_INPUT
  l_i : I1;
END_VAR
```

This input can receive all function blocks that implement interface `I1`.

Example for function calls:
```
DeliverName(l_i := A_instance); // call with instance of
type A
DeliverName(l_i := B_instance); // call with instance of
type B
```

> **NOTE:** A variable of an interface-type must get assigned an instance of a function block before a method can be called on it. A variable of an interface-type always is a reference to the assigned function block instance.

Thus a call to the interface method results in a call to the function block implementation. As soon as the reference is assigned, the corresponding address is monitored in online mode. Otherwise, if no reference has been assigned yet, the value 0 is displayed within monitoring in online mode.

For this example see in the implementation part of the function `DeliverName`:
```
DeliverName := l_i.GetName(); // in this case it depends on
the "real" type of l_i whether A.GetName or B.GetName is
called
```

> **NOTE:** See also the possibility to extend a function block, page 197 by using the keyword `EXTENDS` in the declaration.

Method Invocation

Overview

Object-oriented programming with function blocks is - besides of the possibility of extension, page 197 via `EXTENDS` - supported by the possible use of interfaces, page 198 and inheritance. This requires dynamically resolved method invocations, also called virtual function calls.

Virtual function calls need some more time than normal function calls and are used when:

- a call is performed via a pointer to a function block (`pfub`^.method)
- a method of an interface variable is called (interface1.method)
- a method calls another method of the same function block
- a call is performed via a reference to a function block
- `VAR_IN_OUT` of a base function block type can be assigned an instance of a derived function block type

Virtual function calls make possible that the same call in a program source code will invoke different methods during runtime.

For more information and in-depth view, refer to:

- *Method*, page 181 for further information on methods.
- *THIS Pointer*, page 203 for using THIS pointer.
- *SUPER Pointer*, page 201 for using SUPER pointer.

Calling Methods

According to the IEC 61131-3 standard, methods such as normal functions, page 179 can have additional outputs. They can be assigned in the method call according to syntax:

<method>(in1:=<value> |, further input assignments, out1 => <output variable 1> | out2 => <output variable 2> | ...further output variables)

This has the effect that the output of the method is written to the locally declared variables as given within the call.

Example

Assume that function blocks `fub1` and `fub2` `EXTEND` function block `fubbase` and `IMPLEMENT` `interface1`. Method `method1` is contained.

Possible use of the interfaces and method calls:

```
PROGRAM PLC_PRG
VAR_INPUT
  b : BOOL;
END_VAR
VAR
  pInst : POINTER TO fubbase;
  instBase : fubbase;
  inst1 : fub1;
  inst2 : fub2;
  instRef : REFERENCE to fubbase;
END_VAR
IF b THEN
  instRef REF= inst1;            (* Reference to fub1 *)
  pInst := ADR(instBase);
ELSE
  instRef REF= inst2;            (* Reference to fub2 *)
  pInst := ADR(inst1);
END_IF
pInst^.method1();                 (* If b is true, fubbase.
method1 is called, else fub1.method1 is called *)
instRef.method1();                (* If b is true, fub1.
method1 is called, else fub2.method1 is called *)
```

Assume that `fubbase` of the upper example contains 2 methods `method1` and `method2`. `fub1` overrides `method2` but not `method1`.

`method1` is called as shown in the upper example.

```
pInst^.method1(); (* If b is true fubbase.method1 is called,
else fub1.method1 is called *)
```

For calling via THIS pointer, refer to *THIS Pointer*, page 203.

SUPER Pointer

Overview

For each function block that extends a base function block, a pointer with name `SUPER` is automatically available. It points to the base function block instances.

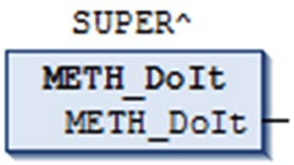
This provides an effective solution for the following issue:

- SUPER offers access to methods of the base function block implementation. With the keyword SUPER, a method can be called which is valid in the base (parent) class instance. Thus, no dynamic name binding takes place.

SUPER may only be used in methods and in the associated function block implementation.

Because SUPER is a pointer to the base function block, you have to dereference it to get the address of the function block: SUPER^.METH_DoIt

SUPER Call in Different Implementation Languages

Implementation Language	Example
ST	`SUPER^.METH_DoIt();`
FBD/CFC/LD	

NOTE: The functionality of SUPER is not yet implemented for Instruction List.

Example

Use of SUPER and THIS pointers.

```
FUNCTION_BLOCK FB_Base
VAR_OUTPUT
    iCnt : INT;
END_VAR
    METHOD METH_DoIt : BOOL
    iCnt := -1;

    METHOD METH_DoAlso : BOOL
    METH_DoAlso := TRUE;

FUNCTION_BLOCK FB_1 EXTENDS FB_Base
VAR_OUTPUT
    iBase: INT;
END_VAR
// Calls the method defined under FB_1
THIS^.METH_DoIt();
THIS^.METH_DoAlso();
// Calls the method defined under FB_Base
SUPER^.METH_DoIt();
SUPER^.METH_DoAlso();
iBase := SUPER^.iCnt;

    METHOD METH_DoIt : BOOL
    iCnt := 1111;
    METH_DoIt := TRUE;

PROGRAM PLC_PRG
VAR
    myBase: FB_Base;
    myFB_1: FB_1;
    iTHIS: INT;
    iBase: INT;
END_VAR
myBase();
iBase := myBase.iCnt;
myFB_1();
```

```
iTHIS := myFB_1.iCnt;
```

THIS Pointer

Overview

For each function block, a pointer with name `THIS` is automatically available. It points to its own function block instance.

This provides an effective solution for the following issues:

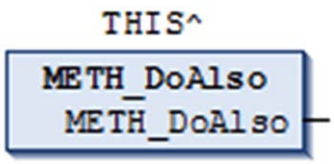
- If a locally declared variable in the method hides a function block variable.
- If you want to refer a pointer to its own function block instance for using in a function.

`THIS` may only be used in methods/properties/actions/transitions and in the associated function block implementation.

`THIS` must be written in capital letters. Other spellings are not accepted.

Because `THIS` is a pointer to the function block, you have to dereference it to access an element of the function block: `THIS^.METHDoIt`.

THIS Call in Different Implementation Languages

Implementation Language	Example
ST	`THIS^.METH_DoIt();`
FBD/CFC/LD	

NOTE: The functionality of `THIS` is not yet implemented for Instruction List.

Example 1

Local variable `iVarB` shadows the function block variable `iVarB`.

```
FUNCTION_BLOCK fbA
VAR_INPUT
    iVarA: INT;
END_VAR
iVarA := 1;

FUNCTION_BLOCK fbB EXTENDS fbA
VAR_INPUT
    iVarB: INT := 0;
END_VAR
iVarA := 11;
iVarB := 2;

    METHOD DoIt : BOOL
    VAR_INPUT
    END_VAR
    VAR
        iVarB: INT;
    END_VAR
    iVarB := 22; // Here the local iVarB is set.
```

```
    THIS^.iVarB := 222; // Here the function block variable
iVarB is set, although iVarB is overloaded.

PROGRAM PLC_PRG
VAR
    MyfbB: fbB;
END_VAR

MyfbB(iVarA:=0 , iVarB:= 0);
MyfbB.DoIt();
```

Example 2

Function call that needs a reference to its own instance.

```
FUNCTION funA
VAR_INPUT
    pFB: fbA;
END_VAR
...;

FUNCTION_BLOCK fbA
VAR_INPUT
    iVarA: INT;
END_VAR
...;

FUNCTION_BLOCK fbB EXTENDS fbA
VAR_INPUT
    iVarB: INT := 0;
END_VAR
iVarA := 11;
iVarB := 2;

    METHOD DoIt : BOOL
    VAR_INPUT
    END_VAR
    VAR
        iVarB: INT;
    END_VAR
    iVarB := 22;    //Here the local iVarB is set.
    funA(pFB := THIS^);    //Here funA is called with THIS^.

PROGRAM PLC_PRG
VAR
    MyfbB: fbB;
END_VAR
MyfbB(iVarA:=0 , iVarB:= 0);
MyfbB.DoIt();
```

Application Objects

Data Type Unit (DUT)

Overview

Along with the standard data types, you can define your own data types. You can create structures, page 661, enumeration types, page 663, and references, page 646 as data type units (DUTs) in a DUT editor, page 381.

For a description of the particular standard and the user-defined data types, refer to the description of the data types, page 635.

Adding a DUT Object

To add a DUT object to an existing application, select the application node in the **Applications tree**, click the green plus button, and select **DUT...**. Or right-click the respective node and execute the command **Add Object > DUT**. To create an application-independent DUT object, select the **Global** node in the **Applications tree**. In the **Add DUT** dialog box, enter a **Name** for the new data type unit, and choose the desired type **Structure**, **Enumeration**, **Alias**, or **Union**.

To DUT objects of type **Enumeration** you can add text list support. For further information, refer to the description of the **Add text list support** command (see Menu Commands, Online Help).

In case of type **Structure**, you can use the principle of inheritance, thus supporting object-oriented programming. Optionally, you can specify that the DUT extends another DUT which is already defined within the project. Therefore, the definitions of the extended DUT will be automatically valid within the current one. For this purpose, activate the option **Extends:** and enter the name of the other DUT.

Click **Add** to confirm the settings. The editor view for the new DUT opens and you can start editing.

Declaring a DUT Object

Syntax

TYPE <identifier> : <DUT components declaration>END_TYPE

The DUT component declaration depends on the type of DUT, for example, a structure, page 661, or an enumeration, page 663.

Example

The following example contains 2 DUTS, defining structures `struct1` and `struct2`; `struct2` extends `struct1`, which means that you can use `struct2.a` in your implementation to access variable `a`.

```
TYPE struct1 :
    STRUCT
        a:INT;
        b:BOOL;
    END_STRUCT
END_TYPE
TYPE struct2 EXTENDS struct1 :
    STRUCT
  c:DWORD;
  d:STRING;
    END_STRUCT
END_TYPE
```

Global Variable List - GVL

Overview

A global variables list (GVL) is used to declare global variables, page 569. If a GVL is placed in the **Applications tree**, the variables will be available for the entire project. If a GVL is assigned to a certain application, the variables will be valid within this application.

To add a GVL to an existing application, select the application node in the **Applications tree**, click the green plus button and select **Global Variable List...**. Alternatively you can right-click the node and execute the command **Add Object >**

Add Global Variable List.... If you select the **Global** node in these views, the new GVL object will application-independent.

Use the GVL editor, page 382 to edit a global variable list.

The variables contained in a GVL can be defined to be available as network variables, page 804 for a broadcast data exchange with other devices in the network. For this purpose, configure network properties (in the menu **View > Properties > Network Variables** or **Network Variable Sender Properties**) for the GVL.

NOTE: The maximum size of a network variable is 255 bytes. The number of network variables is not limited.

NOTE: Variables declared in GVLs get initialized before local variables of POUs.

GVL for Configurable Constants (Parameter List) in Libraries

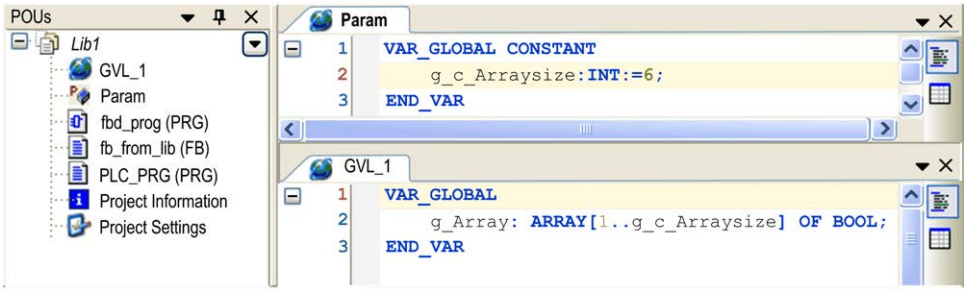
The value of a global constant provided via a library can be replaced by a value defined by the application. For this purpose, the constant has to be declared in a parameter list in the library. Then, when the library is included in the application, its value can be edited in the **Parameter List** tab of the **Library Manager** of the application. See the following example for a description on how to do in detail.

Parameter List Handling

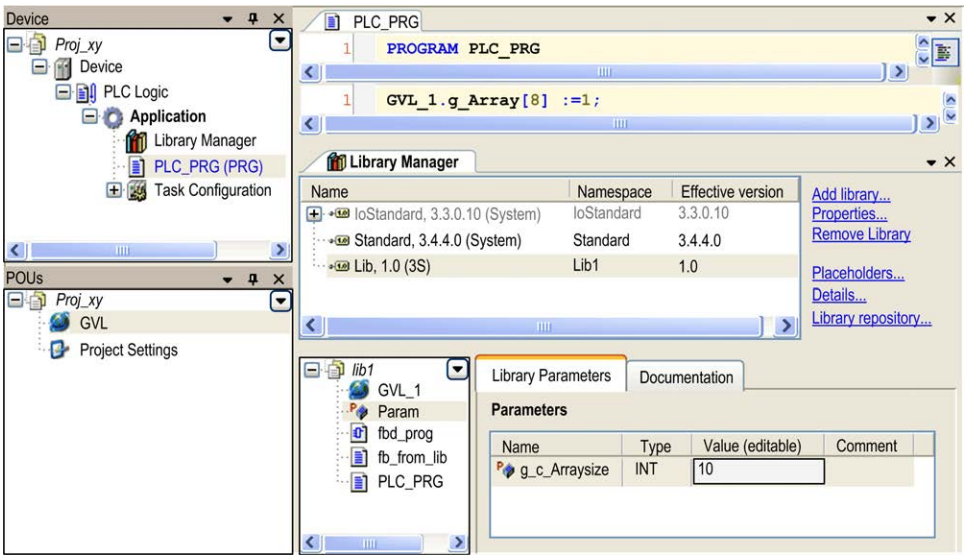
A library `lib1.library` provides an array variable `g_Array`. The size of the array variable is defined by a global constant `g_c_Arraysize`. The library is included in various applications, each needing a different array size. Therefore, you want to overwrite the global constant of the library by an application-specific value.

Proceed as follows: When creating `lib1.library`, define the global constant `g_c_Arraysize` within a special type of global variable list (GVL), the so-called parameter list. For this purpose, execute the command **Add Object** and add a parameter list object, in the current example named `Param`. In the editor of this object, which equals that of a standard GVL, insert the declaration of variable `g_c_Arraysize`.

Parameter list `Param` in library `Lib1.library`



Edit parameter `g_c_Arraysize` in the **Library Manager** of a project



Select the library in the upper part of the **Library Manager** to get the module tree. Select `Param` in order to open the tab **Library Parameters** showing the declarations. Select the cell in column **Value (editable)** and use the empty space to open an edit field. Enter the desired new value for `g_c_Arraysize`. It will be applied to the current, local scope of the library after having closed the edit field.

Global Variable List - GVL (tasklocal)

Overview

The global variables, page 569 declared in a **Global Variable List (tasklocal)** are cycle-consistent. In a task cycle, they are written by one specific task whereas other tasks have read-only access. It is taken into account that tasks can be interrupted by other tasks or can run simultaneously. This helps to provide consistency of variables, even if the application is running on a system with a multicore processor.

In contrast to the common global variable lists that allow multiple tasks to write simultaneously to the GVL variables within one task cycle, the tasklocal GVL provides for automatic synchronization by the compiler when multiple tasks are processing the same variables.

> **NOTE:** As the synchronization of tasklocal variables is time and memory consuming, refer to *Good Practices*, page 210 for further technical information helping you to evaluate whether this solution fits to your application needs.

The object node **Global Variable List (tasklocal)** represents tasklocal global variable lists. In addition to a common GVL object, it indicates the task that has exclusive write access to the variables within a task cycle.

> **NOTE:** The values of tasklocal global variables cannot be modified in online mode with the **Write Values** command.

Sample Application

The following sample application indicates the principle and functionality of tasklocal variables. It includes a writing program (*WriteData*) and a reading program (*ReadData)*. The programs run in different tasks but they access the same data. This data is stored in a **Global Variable List (tasklocal)** to allow for a cycle-consistent processing. Also refer to the paragraph *Instructions for Creating the Sample Application*, page 211.

```
(* task-local GVL, object name: "Tasklocals" *)
VAR_GLOBAL
    g_diaData : ARRAY [0..99] OF DINT;
END_VAR

PROGRAM ReadData
VAR
    diIndex : DINT;
    bTest : BOOL;
    diValue : DINT;
END_VAR
bTest := TRUE;
diValue := TaskLocals.g_diaData[0];
FOR diIndex := 0 TO 99 DO
    bTest := bTest AND (diValue = Tasklocals.g_diaData
[diIndex]);
END_FOR

PROGRAM WriteData
VAR
    diIndex : DINT;
    diCounter : DINT;
END_VAR
diCounter := diCounter + 1;
FOR diIndex := 0 TO 99 DO
    Tasklocals.g_diaData[diIndex] := diCounter;
END_FOR
```

In the program *WriteData*, the array *g_diaData* is populated with values. The validity of these values is verified by the program *ReadData*. If the values are valid, the variable *bTest* is set to TRUE.

The array data is declared with the variable *g_diaData* in the object *Tasklocals* of type *Global Variable List (tasklocal)*. This helps the compiler to manage the synchronization of data access by the different tasks and helps to keep cycle consistency even when the accessing programs are called from different tasks. In the sample application, this has the effect that the variable *bTest* is constantly TRUE in the program *ReadData*.

> **NOTE:** If the variable *g_diaData* was declared as a common global variable instead, the variable *bTest* would be set to FALSE whenever one of the two tasks in the `FOR` loop is interrupted by the other task or both tasks run simultaneously (multicore controllers). This would not prevent the writer from changing the values while the reader is reading the list.

Constraints for the Declaration

When you declare a **Global Variable List (tasklocal)**, consider the following:

- Do not assign direct addresses using an AT declaration.
- Do not map tasklocal variables in the controller configuration.
- Do not declare pointers.
- Do not declare references.
- Do not instantiate function blocks.
- Do not declare tasklocal variables as both `PERSISTENT` and `RETAIN` at the same time.

  NOTE: You cannot perform an online change of the application after you have modified the declarations in a **Global Variable List (tasklocal)**.

Write attempts of tasks without write access are detected as an error by the compiler. However, as the compiler can only assign static calls to a task, not all write-access violations are detected. The call of a function block by using a pointer or an interface, for example, is not assigned to a task and is thus not detected. Moreover, if pointers point to tasklocal variables, data can be modified in a read task. This will not be detected as a runtime error. However, values that are modified by using pointer access are not copied back to the shared reference of variables.

Advantages and Constraints of a Global Variable List (tasklocal)

The variables are located at a different address in the **Global Variable List (tasklocal)** for each task. For read access, this has the effect that `ADR (variable name)` yields a different address in each task.

The synchronization mechanism provides the following advantages:

- Cycle-consistent processing.
- No locked states: A task is not waiting for an action to be completed by another task.

Nevertheless, the following constraints apply:

- It is not possible to determine a time when a reading task will by all means receive a 1:1 copy of the writing task. In the example above, you cannot assume that each written copy is processed once by the reader. The copies can deviate because the reading task can edit the same array over multiple cycles, or the contents of the array can skip one or more values between two cycles. Both scenarios can occur and must be considered.
- Access to the shared variables of the writing task can be inhibited by each reading task for one cycle. This can have the effect that n reading tasks lead to a delay of n cycles of the writing task until the next update of the shared reference.
- In each task cycle, the writing task can prevent a reading task from getting a reading copy. Thus, it is not possible to specify a maximum number of cycles after which a reading task will definitely receive a copy.

This has to be considered, in particular, if slow running tasks are involved. Assuming that a task is running once an hour and is prevented from accessing the tasklocal variables, then the data it is processing is considerably outdated.

For the reading task to determine whether the list is up-to-date, you can add a time stamp to the tasklocal variables as follows:

1. Add a variable `g_timestamp` of type LTIME to the list of tasklocal variables.
2. Add the following code to the writing task:
   ```
   tasklocal.g_timestamp := LTIME();
   ```

Good Practices

Tasklocal variables are designed for the use case of a single writer and multiple readers.

When you implement program code that is called by different tasks, tasklocal variables provide the following advantages:

- As indicated by the sample application, page 208, the application `appTasklocal` is extended by multiple reading tasks that access the same array and use the same functions.
- On multicore systems, you cannot synchronize tasks by priority. The tasklocal variables provide an alternative synchronization mechanism.

For the following use cases, tasklocal variables must not be used:

- When it is essential that a reading task is supplied with the newest copy of the variables.
- When a task produces data and another task processes the data (also known as Producer - Consumer dilemma).

  Select another type of synchronization for this configuration. For example, the producer can insert a flag to indicate that updated data is available. The consumer can insert a second flag to indicate that the data has been processed and is ready to receive new input. This has the advantage that both can work on the same data. The overhead for cyclic data copying is avoided. The consumer does not lose data generated by the producer.

Monitoring

At runtime, it can happen that different copies of the **Global Variable List (tasklocal)** are available in the memory. When monitoring a position, not all available values can be displayed. For inline monitoring, the values of the shared reference are displayed in the watch list and in the visualization for a tasklocal variable.

For breakpoints, consider the following: When a task hits a breakpoint, the task is halted and the data of this task is displayed. In the meantime, the other tasks continue running which may change the shared copy. In the context of the halted task, however, the values remain unchanged and may therefore deviate from the updated values.

Background: Technical Implementation

For a **Global Variable List (tasklocal)**, the compiler creates a copy for each task, as well as a shared reference copy that is available for all tasks. Whenever a variable in the list is accessed from the code, the access refers to the tasklocal copy of the list.

At runtime, at the end of the writing task, the contents of the tasklocal list are written to the global list. At the beginning of a reading task, the contents of the shared reference global list are copied to the tasklocal copy.

The behavior is based on the following task synchronization strategy:

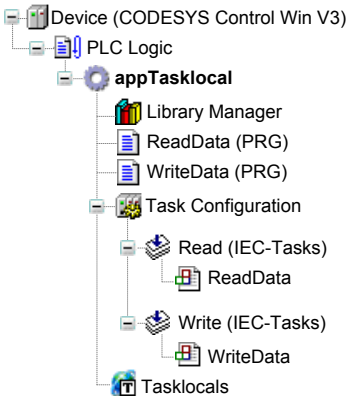
- As long as the writing task is writing a copy back to the shared reference, none of the reading tasks retrieves a copy.
- As long as a reading task is retrieving a copy from the shared reference, the writing task does not write back a copy.

Instructions for Creating the Sample Application

The sample application contains a program `ReadData` that should access the same data that is written by the program `WriteData`. Both programs are running in different tasks. Data is provided in a **Global Variable List (tasklocal)** for being processed in a cycle-consistent manner.

As a prerequisite, create a new default project and open it in the editor.

Step	Action	Result
1	Rename the application node from **Application** to **appTasklocal**.	–
2	Below the **appTasklocal** node, add a program in ST named `ReadData`.	–
3	Below the **appTasklocal** node, add another program in ST named `WriteData`.	–
4	Below the **Task Configuration** node, rename the default task from **MainTask** to `Read`.	–
5	In the **Configuration** dialog box of the `Read` task, click the **Add Call** button to call the program `ReadData`.	–
6	Below the **Task Configuration** node, add another task named `Write` and add the call of the program `Write` to this task.	**Result**: Two tasks `Read` and `Write` are available as subnodes of the **Task Configuration** node. They call the programs `ReadData` and `WriteData`.
7	Select the **appTasklocal** node and add an object of type **Global Variable List (tasklocal)**.	**Result**: The **Add Global Variable List (tasklocal)** dialog box opens.
8	Enter the **Name** `Tasklocals`.	–
9	Select the `Write` task from the **Task with write access** list.	**Result**: The tree structure for using tasklocal variables within an application is complete. Program the objects as described in the sample application, page 208.



Network Variable List (Receiver)

Overview

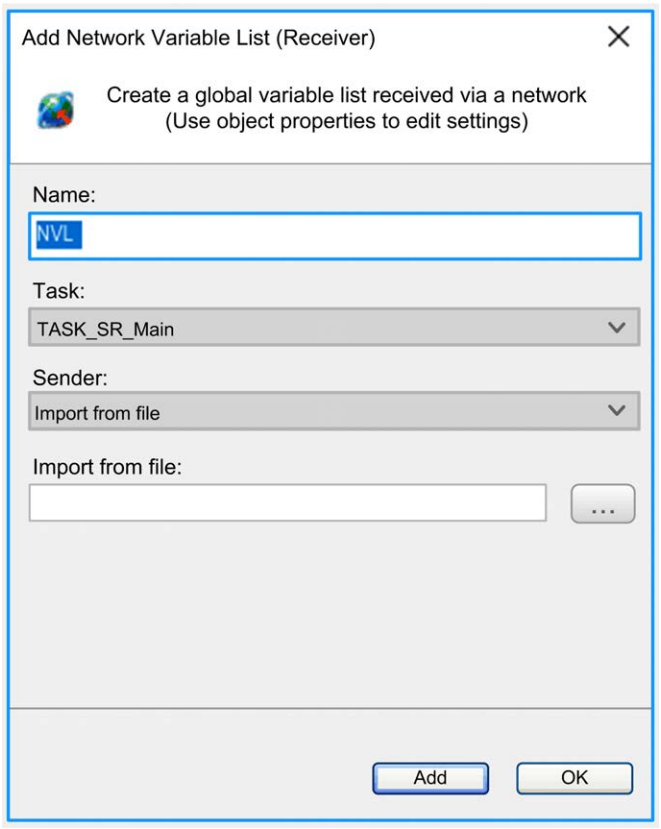
A network variable list (receiver) is added to the **Applications tree**. It defines variables, which are specified as network variables in another device within the network.

NOTE: Consider the Network Variables List (NVL) Rules, page 394.

Thus you can add a network variable list (receiver) object to an application if a network variable list (sender) with special network properties (network variable list) is available in one of the other network devices. This is independent of whether defined in the same project or in different projects. If several of appropriate network variable lists (sender) are found within the present project for the present network, select the desired network variable list (sender) from a selection list **Sender** when adding a network variable list (receiver) via the dialog box **Add Object > Network Variable List (Receiver)**. Network variable lists (sender) from other projects must be imported as described in this chapter.

Therefore, each network variable list (receiver) corresponds exactly to one network variable list (sender) in another device.

Dialog box **Add Network Variable List (Receiver)**



Description of the Elements

When adding the network variable list (receiver), besides a **Name**, also define a **Task**, responsible for the handling of the network variables.

Alternatively to directly choosing a network variable list (sender) from another device, you can specify a network variable list (sender) export file *.GVL with the option **Import from file**. This network variable list (sender) file has been generated previously from the network variable list (sender) via **View > Properties > Link To File** dialog box (see Menu Commands, Online Help). In any case this is necessary if the desired network variable list (sender) is defined within another project. For this purpose, select the option **Import from file** in the **Sender** selection list and enter the file path in the **Import from file** text field (or click the **...** button to open the dialog for browsing in the file system).

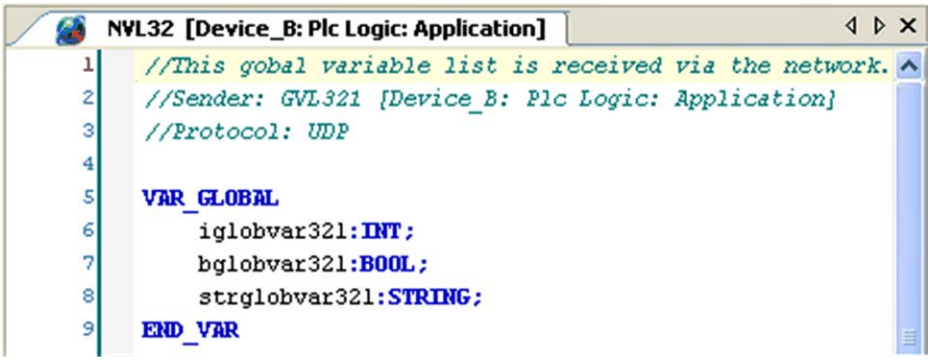
You can modify the settings at a later time via the **View > Properties > Network Settings** dialog box (see Menu Commands, Online Help).

A network variable list (receiver) is displayed by the NVL editor, page 386, but it cannot be modified. It displays the content of the corresponding network variable list (sender). If you modify the basic network variable list (sender), the network variable list (receiver) is updated accordingly.

A comment is added automatically at top of the declaration part of a network variable list (receiver), providing information on the sender (device path), the network variable list (sender) name, and the protocol type.

Network Variable List Example

Network variable list



NOTE: Only arrays whose bounds are defined by a literal or a constant are transferred to the remote application. Constant expressions in this case are not allowed for bounds definition. Example: `arrVar : ARRAY[0..g_iArraySize-1] OF INT ;` is not transferred `arrVar : ARRAY[0..10] OF INT ;` is transferred

For further information, refer to the *Network Communication* chapter, page 804.

Example of a Simple Network Variable Exchange

In the following example, a simple network variable exchange is established. In the sender controller, a network variable list (sender) is created. In the receiver controller, the corresponding network variable list (receiver) is created.

Perform the following preparations in a default project, where a sender controller **Dev_Sender** and a receiver controller **Dev_Receiver** are available in the **Devices tree**:

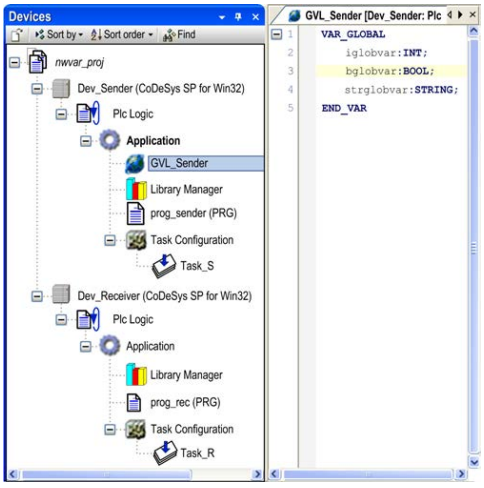
- Create a POU (program) **prog_sender** below the **Application** node of **Dev_Sender**.
- Under the **Task Configuration** node of this application, add the task **Task_S** that calls **prog_sender**.
- Create a POU (program) **prog_rec** below the **Application** node of **Dev_Receiver**.
- Under the **Task Configuration** node of this application, add the task **Task_R** that calls **prog_rec**.

NOTE: The 2 controllers must be configured in the same subnet of the Ethernet network.

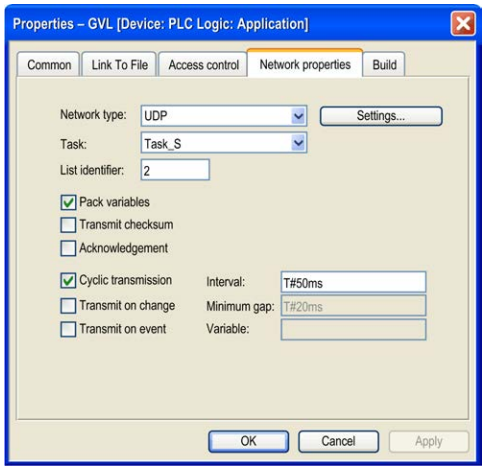
Defining the Network Variable List (Sender)

Step 1: Define a global variable list in the sender controller:

Step	Action	Comment
1	In the **Applications tree**, select the **Application** node of the controller **Dev_Sender** and click the green plus button. Execute the command **Add other objects > Network Variable List (Sender)**.	The **Properties** dialog box of the network variable list (sender) is displayed.
2	Enter the **Name** `GVL_Sender` and click **Add** to create a new global variable list.	The **GVL_Sender** node appears below the **Application** node in the **Applications tree** and the editor opens on the middle of the screen.

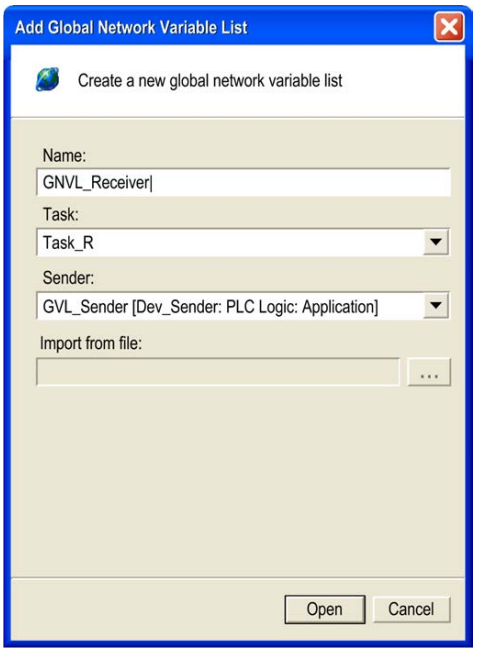
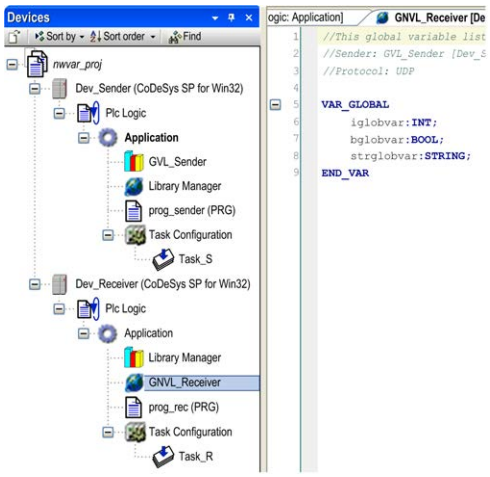
3	In the editor, enter the following variable definitions:
	`VAR_GLOBAL`
	`iglobvar:INT;`
	`bglobvar:BOOL;`
	`strglobvar:STRING;`
	`END_VAR`
	
		–

Step 2: Define the network properties of the network variable list (sender):

Step	Action	Comment
1	In the **Applications tree**, select the **GVL_Sender** node, click the green plus button, and execute the command **Properties...**.	The **Properties - GVL_Sender** dialog box is displayed.
2	Open the **Network properties** tab and configure the parameters as indicated in the graphic: 	–
3	Click **OK**.	The dialog box is closed and the network variable list (sender) network properties are set.

Defining the Network Variable List (Receiver)

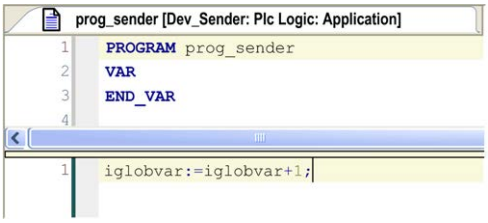
Step 1: Define a global network variable list in the receiver controller:

Step	Action	Comment
1	In the **Applications tree**, select the **Application** node of the controller **Dev_Receiver**, click the green plus button, and execute the command **Global Network Variable List...**.	The **Add Global Network Variable List** dialog box is displayed.
2	Configure the parameters as indicated in the graphic. 	This global network variable list is the counterpart of the network variable list (sender) defined for the sender controller.
3	Click **Open**.	The dialog box is closed and the **GNVL_Receiver** appears below the **Application** node of the **Dev_Receiver** controller:  This network variable list (receiver) automatically contains the same variable declarations as the **GVL_Sender**.

Step 2: View and / or modify the network settings of the network variable list (receiver):

Step	Action	Comment
1	In the **Devices tree**, right-click the **GNVL_Receiver** node and select the command **Properties...**.	The **Properties - GNVL_Receiver** dialog box is displayed.
2	Open the **Network settings** tab.	–

Step 3: Test the network variable exchange in online mode:

Step	Action	Comment
1	Under the **Application** node of the controller **Dev_Sender**, double-click the POU **prog_sender**.	The editor for **prog_sender** opens on the right-hand side.
2	Enter the following code for the variable `iglobvar`: 	–
3	Under the **Application** node of the controller **Dev_Receiver**, double-click the POU **prog_rec**.	The editor for **prog_rec** opens on the right-hand side.
4	Enter the following code for the variable `ivar_local`: 	–
5	Log on with sender and receiver applications within the same network and start the applications.	The variable `ivar_local` in the receiver gets the values of `iglobvar` as currently displayed in the sender.

Persistent Variables

Overview

This object is a global variable list (GVL), which only contains persistent variables of an application. Thus it has to be assigned to an application. For this purpose, it has to be inserted in the **Applications tree** via selecting the respective node, clicking the green plus button, and selecting **Add Other Objects > Persistent Variables...**.

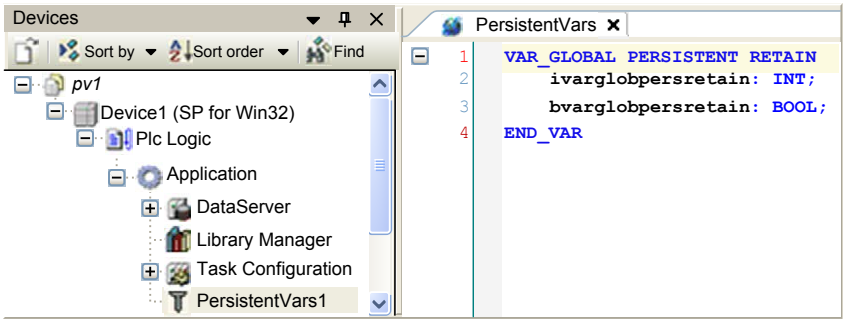
Consult the *Programming Guide* specific to your controller for information on the behavior of remanent variables.

Only those variables which are declared with `VAR PERSISTENT` and which are contained in this list are persistent. The **Add all Instance Paths** command (see Menu Commands, Online Help) adds `PERSISTENT` declarations found in other POUs to the list.

> **NOTE:** When a variable is added by using the command **Add all Instance Paths**, it consumes memory twice and requires more cycle time. Therefore, it is a good practice to declare persistent variables in the persistent variable list only.

Variables declared with `VAR PERSISTENT` are also retain variables. Retain variables have the capacity to keep their values after executing a **Reset warm** command (see Menu Commands, Online Help). The difference is, that persistent variables are only reinitialized upon executing the **Reset origin** command (see Menu Commands, Online Help) or by a new application download (after the application had been removed from the controller). An exception is made if you modified their names or data types.

Persistent variable list



For further information, refer to the description of remanent variables, page 571.

Also refer to the description of the special commands for handling persistent variables (see Menu Commands, Online Help).

Edit a persistent variable list in the persistence editor, which corresponds to the GVL editor, page 382. The `VAR_GLOBAL PERSISTENT RETAIN` is already preset in the first line.

Adding and Declaring Remanent Variables

When you add variables to an application, you can declare some of the variables as remanent variables. Remanent variables can retain their values in the event of power outages, reboots, resets, and application program downloads. There are multiple types of remanent variables, declared individually as retain or persistent, or in combination as retain-persistent.

Consult the *Programming Guide* specific to your controller for information on the memory size reserved for retain and persistent variables in the different controllers.

To add a global variable list called **Persistent Variables** to your application, proceed as follows:

Step	Action
1	Select the respective application node in the **Applications tree**, click the green plus button, and select **Add Other Objects > Persistent Variables...**. Alternatively, you can right-click the application node, and execute the command **Add Object > Persistent Variables...**.
2	In the **Add Persistent Variables** dialog box type a name for this list in the **Name** text box.
3	Click **Add**. **Result:** A persistent variable node is created in the **Applications tree**. For an example, refer to the *Overview* paragraph in this chapter.

External File

Overview

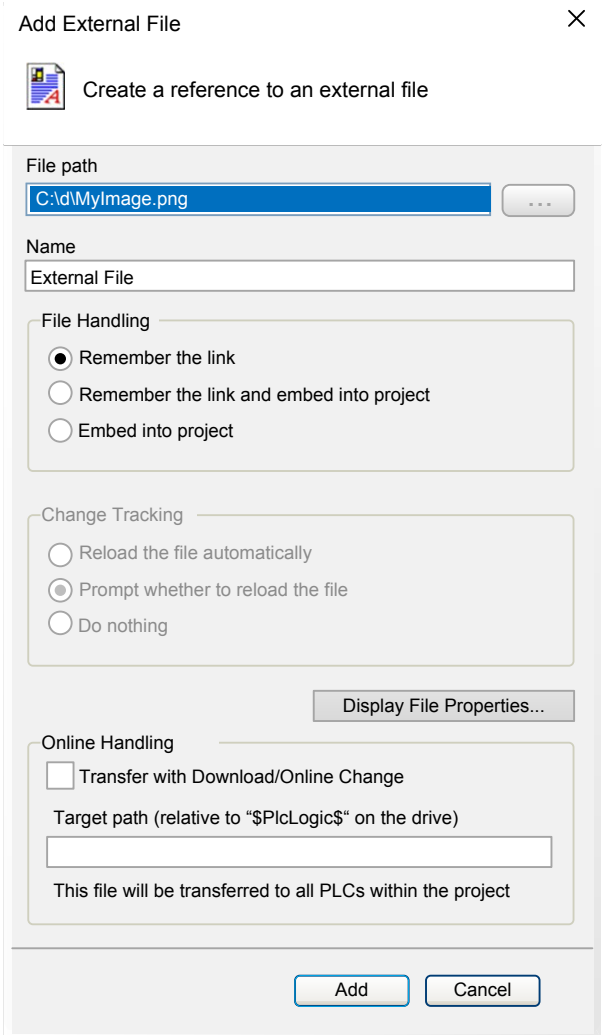
To add an external file to the **Global** node of the **Applications Tree** or **Tools Tree**, select the **Global** node, click the green plus button and execute the commands **Add other objects > External File...**.

> **NOTE:** External files which are inserted in the **Global** node of the **Applications tree** or **Tools tree** are transferred to all controllers in the project during the download. This option has no effect on external files in libraries. These files are not transferred.

An external file inserted in the **Devices tree** is downloaded to the controller when an online change or a download is performed.

Click the **...** button to open the dialog box for browsing a file. The path of this file is entered in the **File path** text box. In the **Name** text box, the name of the chosen file is entered automatically without extension. You can edit this field to define another name for the file under which it should be handled within the project.

Add External File dialog box:



Description of the File Handling Section of the Dialog Box

Select one of the following options:

Option	Description
Remember the link	The file will be available in the project only if it is available in the defined link path
Remember the link and embed into project	A copy of the file will be stored internally in the project but also the link to the external file will be recalled. As long as the external file is available as defined, the defined update options will be implemented accordingly. Otherwise just the file version stored in the project will be available.
Embed into project	Just a copy of the file will be stored in the project. There will be no further connection to the external file. When you open the external file (using the **Edit > Object** command), a temporary file is created for editing.

Description of the Change Tracking Section of the Dialog Box

If the external file is linked to the project, you can additionally select one of the options:

Option	Description
Reload the file automatically	The file is updated within the project as soon as it has been changed externally.
Prompt whether to reload the file	A dialog box pops up as soon as the file has been changed externally. You can decide whether the file is updated also within the project.
Do nothing	The file remains unchanged within the project, even when it is changed externally.

Description of the Online Handling Section of the Dialog Box

Option	Description
Transfer with Download/ Online Change	If selected, the external file is stored in the folder specified with **Target path (relative to "$PlcLogic$" on the device)** after download / online change. **NOTE:** `$PlcLogic$` is a placeholder for the folder on the controller that contains the application. **NOTE:** External files that are inserted in the **Global** node of the **Applications Tree** or **Tools Tree** are transferred to all controllers in the project during the download. This option has no effect on external files in libraries. These files are not transferred.
Target path (relative to "$PlcLogic$" on the device)	You can specify the target path as follows: • For the `$PlcLogic$` root directory: Leave the input field blank. • Folder of the application (below `$PlcLogic$`) Example for the *"App123"* application: `App123` • Nested folder structure below the application folder Example: `App123/Sub01/SubSub01` • Using another available placeholder Example for the visualization: `$visu$` **NOTE:** The paths are case-sensitive. **NOTE:** `$PlcLogic$` is a placeholder for the folder on the controller that contains the application.

Description of the Buttons

Button	Description
Display File Properties...	This button opens the dialog box for the properties of a file. This dialog box also appears when you select the file object in the **Applications Tree** or **Tools Tree** and execute the command **Properties**. In the tab **External file** of this dialog box, you can display and modify the properties (also refer to the Menu Commands Online Help).
Add	After you have completed the settings, click the **Add** button to add the file to the **Global** node of the **Applications Tree** or **Tools Tree**. It is opened in the tool which is defined as default for the given file format.

Text List

Overview

A text list is an object managed globally in the **Global** node of the **Applications Tree** or assigned to an application in the **Applications Tree**.

It serves the following purposes:

- Multi-language support for static, page 223 and dynamic, page 224 texts and tooltips in visualizations and in the alarm handling
- Dynamic text exchange

Text lists can be exported and (re-) imported, page 226. Export is necessary, if a language file in XML format has to be provided for a target visualization, but is also useful for translations, page 227.

Possible formats of text lists:

- Text
- XML

You can activate support of Unicode, page 226.

Each text list is uniquely defined by its namespace. It contains text strings which are uniquely referenced within the list by an identifier (ID, consisting of any sequence of characters) and a language identifier. The text list to be used is specified when configuring the text for a visualization element.

Depending on the language which is set in the visualization, the corresponding text string is displayed in online mode. The language used in a visualization is changed by a **Change the language** input. This is accomplished by a mouse action that you have configured on the given visualization element. Each text list must at least contain a default language, and optionally in other languages that you choose to define. If no entry is found which matches the set language, the default language entry of the text list is used. Each text can contain formatting definitions, page 227.

Basic structure of a text list

Identifier (Index)	Default	<Language 1>	<Language 2>	.... <Language n>
<unique string of characters>	<text abc in default language>	<text abc in language 1>	<text abc in language 2>	...
<unique string of characters>	<text xyz in default language>	<text xyz in language 1>	<text xyz in language 2>	...

Text List Types

There are two types of text usable in visualization elements and correspondingly there are two types of list:

- `GlobalTextList` for static texts
- Textlist for dynamic texts

GlobalTextList for Static Texts

GlobalTextList is a special text list where the identifiers for the particular text entries are handled implicitly and are not editable. The list can be exported, edited externally and then reimported.

Static texts in a visualization, in contrast to dynamic texts, are not exchanged by a variable in online mode. The only option to exchange the language of a visualization element is via a **Change the language** input. A static text is assigned to a visualization element via property **Text** or **Tooltip** in category **Texts**. When the first static text is defined in a project, a text list object named **GlobalTextList** is added to the **Global** node of the **Applications Tree**. It contains the defined text string found in the column **Default**, and an automatically assigned integer number as the text identifier. For each static text that is created thereafter, the identifier number is incremented and assigned to the visualization element.

If a static text is entered into a visualization element (for example, if in a rectangle with property category of **Texts**, the string **Text Example** is specified), this text is looked up in the **GlobalTextList**.

- If the text is found (for example, **ID 4711**, **Text Example**), the element value **4711** of **TextId** will be assigned to an internal variable. This establishes the relationship between the element and the corresponding line in the **GlobalTextList**.
- If the text is not found, a new line is inserted in the **GlobalTextList** (for example, **ID 4712, Text Example**). In the element, the value **4712** is assigned to the internal variable.

  NOTE: If it does not yet exist - you can create a global text list explicitly by the command **Create Global Text List**.

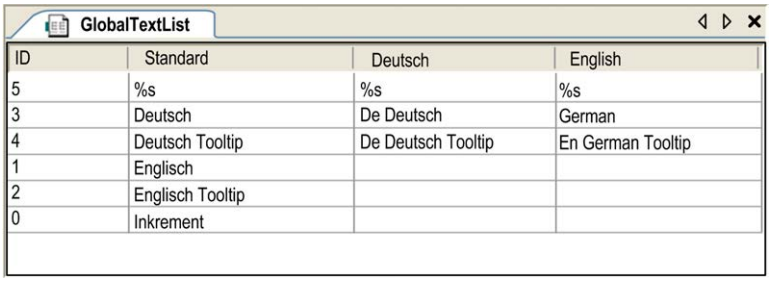
If you have exported, edited and reimported the **GlobalTextList**, it is validated as to whether the identifiers are still matching those which are used in the configuration of the respective visualization elements. If necessary, an implicit update of the identifiers used in the configuration will be implemented.

To update the ID numbers of the texts that are defined for a visualization, you can remove the **GlobalTextList** by right-clicking the node and executing the **Delete** command. Then open a visualization and execute the command **Visualization > Create Global Text List**. A new **GlobalTextList** node is created in the **Applications tree** with the static texts of the visualizations available in the project.

NOTE: In case your **GlobalTextList** contains translated strings, they will not be regenerated when the command **Create Global Text List** is executed.

Example of a GlobalTextList

Create Global Text List



ID	Standard	Deutsch	English
5	%s	%s	%s
3	Deutsch	De Deutsch	German
4	Deutsch Tooltip	De Deutsch Tooltip	En German Tooltip
1	Englisch		
2	Englisch Tooltip		
0	Inkrement		

Textlist for Dynamic Texts

Dynamic texts can be modified dynamically in online mode. The text index (**ID**), which is a string of characters, must be unique within the text list. In contrast to **GlobalTextLists**, you have to define it. Also in contrast to the **GlobalTextList**, create text lists for dynamic texts explicitly by selecting the **Global** node, clicking the green plus button, and executing the command **Add other objects > Text List...**.

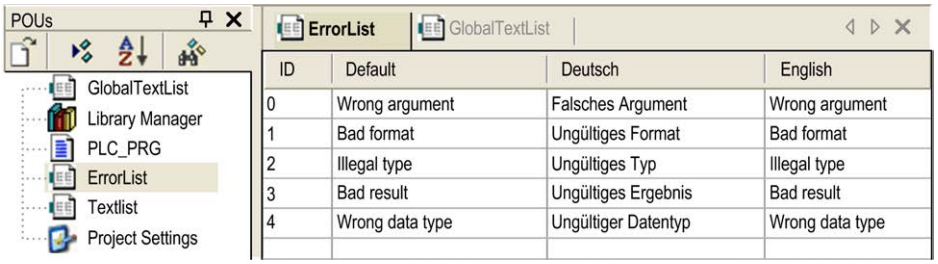
The available dynamic text lists are offered when configuring a visualization element via property **Dynamic texts** / **Text list**. If you specify a text list name combined with the text index (**ID**) - which can be entered directly or by entering a project variable which defines the ID string - the text can be modified in online mode.

A dynamic text list must be exported if it is needed as a language file for language switching in a target visualization. Specify the file path in the **Visualization Options**. Such as **GlobalTextList**, a dynamic text list can also be exported for external editing and reimported. In contrast to **GlobalTextList**, when you import dynamic text lists, there is no automatic check and update of the identifiers.

NOTICE
UNINTENDED MODIFICATION OF IDENTIFIERS Do not modify the identifiers when editing the exported list. **Failure to follow these instructions can result in equipment damage.**

Example of a Dynamic Text List Named ErrorList

Example **ErrorList**



ID	Default	Deutsch	English
0	Wrong argument	Falsches Argument	Wrong argument
1	Bad format	Ungültiges Format	Bad format
2	Illegal type	Ungültiges Typ	Illegal type
3	Bad result	Ungültiges Ergebnis	Bad result
4	Wrong data type	Ungültiger Datentyp	Wrong data type

Detailed Example

This example explains how to configure a visualization element, which displays the corresponding message when an error is detected in an application that processes error events identified via numeric IDs assigned to an integer variable `ivar_err`.

Provide a dynamic textlist named **ErrorList** where the message texts for error IDs 0 to 4 are defined in languages **German**, **English**, and **Default**:



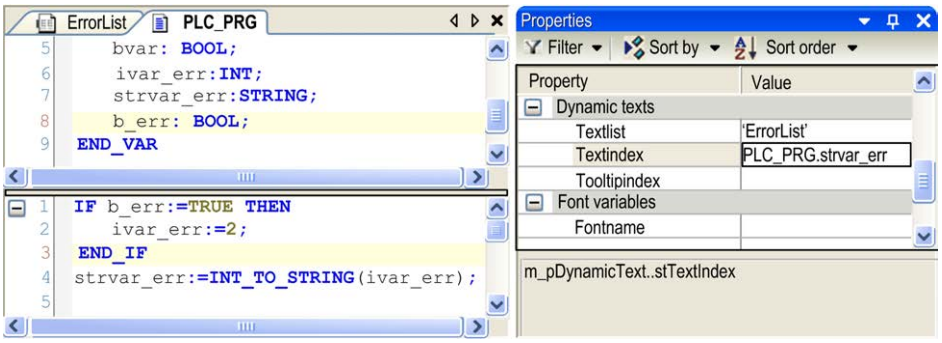
ID	Default	Deutsch	English
0	This is Error 0. Do the following...	Fehler 0. Führen Sie folge...	Error 0. Do the...
1	This is Error 1. Close...	Fehler 1. Schließen Sie...	Error 1. Close the...
2	This is Error 2. Perform a...	Fehler 2. Führen Sie einen...	Error 2. Perform...
3	This is Error 3. Try to...	Fehler 3. Versuchen Sie...	
4	This is Error 4. Start...	Fehler 4. Starten Sie...	

Within a table cell, you can add a line break by pressing the keyboard shortcut **Ctrl** + **Enter**.

To use the error IDs in the visualization configuration, define a STRING variable, for example `strvar_err`. To assign the integer value of `ivar_err` to `strvar_err`, use `strvar_err:=INT_TO_STRING(ivar_err);`.

`strvar_err` can be entered as **Textindex** parameter in the configuration of the **Dynamic texts** properties of a visualization element. This element will display the appropriate message in online mode.

The next example is for processing the error ID using project variables and configuration of a visualization element (**Properties**), which should display the appropriate message:



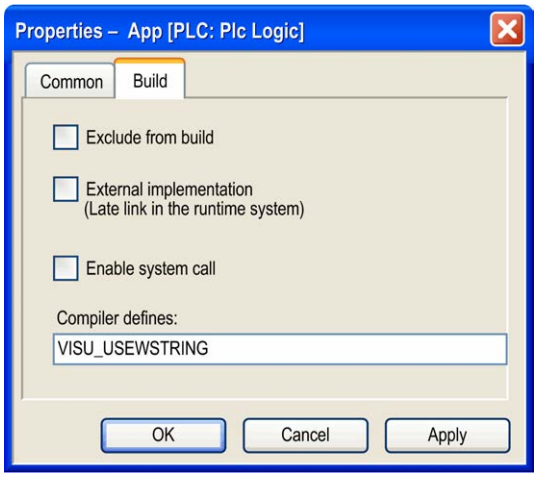
Creating a Text List

- To create a text list for dynamic texts, page 224, add a **Text List** object to the project in the **Applications Tree**. To create an application-specific text list, select an application node. To create a global text list, select the **Global** node. Then click the green plus button of the selected node, and execute the command **Add other objects > Text List...**. When you have specified a list name and confirmed the **Add Textlist** dialog box, the new list is inserted below the selected node, and a text list editor view opens.
- To get a text list for static texts, page 223 (**GlobalTextList**), either assign a text in property **Text** in category **Texts** of a visualization object to get the list created automatically, or generate it explicitly by command **Create Global Text List**.
- To open an existing text list for editing, select the list object in the **Applications Tree** or **Global** node of the **Applications Tree**. Right-click the text list node and execute the command **Edit Object**, or double-click the text list node. Refer to the table *Basic structure of a text list* for how a text list is structured.
- For adding a new default text in a text list, either use the command **Insert Text**, or edit the respective field in the empty line of the list. To edit a field in a text list, click the field to select it and then click the field again or press SPACE to get an edit frame. Enter the desired characters and close the edit frame with RETURN.

Support of Unicode Format

To use Unicode format, activate the respective option in the **Visualization Manager**. Further on, set a special compilation directive for the application: select the application in the **Devices Tree**, open the **Properties** dialog box, **Build** tab. In the **Compiler defines** field, enter `VISU_USEWSTRING`.

Dialog box with compiler definition



Export and Import of Text Lists

Static and dynamic text lists can be exported as files in CSV format. Exported files can also be used for adding texts externally, for example by an external translator. However, only files available in text format (**.csv*) can be reimported.

See the description of the respective text list commands (see Menu Commands, Online Help).

Specify the folder in which the export files should be saved in the dialog box **File > Project Settings > Visualization**.

Formatting of Texts

The texts can contain formatting definitions (`%s`, `%d`,…), which allow to include the present values of variables in a text. For the possible formatting strings, see the *Visualization* part of the online help.

When using text with formatting strings, the replacement is done in the following order:

- The actual text string to be used is searched via list name and ID.
- If the text contains formatting definitions, these are replaced by the value of the respective variable.

Subsequent Delivery of Translated Texts

By inserting *GlobalTextList.csv* in the directory which is used for loading text files, a subsequent integration of translated texts is possible. When the bootproject is started up, the firmware detects that an additional file is available. The text is compared with that in the existing textlist files. New and modified texts are then applied to the textlist files. The updated textlist files will then be applied at the next startup.

List Components for Text Input

Use the dialog box **Tools > Options > Visualization**, to specify a text template file. All texts of column **Default** of this file will be copied to a list, which will be used for the **List Components** functionality. A template file can be used which has been created before via the **Export** command.

Multiple User Operations

By use of the source control, it is possible that multiple users work simultaneously on the same project. If a static text is modified in visualization elements by more than one user, it will cause modifications to the **GlobalTextList** (refer to **GlobalTextList**, page 223). In this case, the Text-Ids may no longer be coherent with the visualization elements. Use the following error detection and correction methods:

- Use the command **Check Visualization Text Ids**, such errors may be detected in the visualizations.
- Use the command **Update Visualization Text Ids**, these errors may be resolved automatically. The affected visualizations as well as the **GlobalTextList** must have write permission.

Use of Textlists for Changing Language in Visualizations

If an appropriate textlist is available, that is, a textlist defining several language versions for a text, then the language used for the texts in a visualization can be switched in online mode by an input on a visualization element. The **Dynamic Texts** properties of the element must specify the textlist to be used, and an **OnMouse..** input action, **Change the language**, must be configured specifying the language which should be used after the mouse action has been performed.

NOTE: The language must be specified with exactly this string which is displayed in the column header of the respective textlist.

Image Pool

Overview

Image pools are tables defining the file path, a preview, and a string ID for each image. By specifying the ID and (for unique accessing) additionally the image file name, the image can be referenced, for example, when being inserted in a visualization (configuration of the properties of an image element, refer to *Using Images Which are Managed in Image Pools*, page 231).

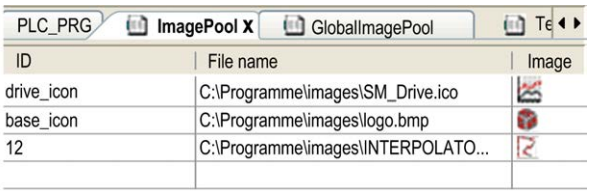
In a library project, you can create an image pool. If you then declare the library to be a symbol library, you can use the images it contains inside your project visualizations. To achieve this, right-click the **ImagePool** node of the library project, select **Properties**, and set the **Symbol library settings** in the **Image Pool** tab (by clicking the button **Mark library as symbol library** and optionally selecting a **Text list for symbol translation**).

After you have added the library to your project, the image pool will appear (as you named it) in the **ToolBox** when a visualization editor is active.

> **NOTE:** Reduce the size of an image file as much as possible before adding it to an image pool. Otherwise, the project size and the loading and storing efforts of visualization applications, including images, can become large.

Structure of an Image Pool

Example of an image pool:



Element	Description
ID	String ID (for example **logo**, **y_icon**, **2**); A unique referencing of an image is achieved by the combination of image list name and ID (for example, **List1.basic_logo**).
File name	Path of the image file (for example, *C:\programs\images\logo.bmp*). Supported image formats: • BMP • EMF • GIF • ICO • JPG • PNG • SVG • TIFF The controller you are using may not support all image formats. Consult the *Programming Guide* specific to your controller for further information. If the image file is stored in the directory for image files (as defined in **Tools > Options > Visualization**, you only have to enter the file name in this text box.
Image	Preview of the image.
Link type	Information on how the image file is linked to the project. Specify the **Link type** when you add the image file manually in the dialog box **Select image**. Refer to the description *Creating and Editing an Image Pool*, page 230.

NOTE: If the target system does not support images in the vector image format SVG, they are automatically converted to the format PNG during download. For information on the supported image formats, refer to the device descriptions provided by your hardware manufacturer.

Creating and Editing an Image Pool

A project can contain several image pools. The automatically generated **GlobalImagePool**, as well as manually generated image pools.

GlobalImagePool

Add an image, which is not yet part of an image pool of the project, to a visualization. In doing so, enter in the element properties a static ID for the image. This results in the automatic creation of a **GlobalImagePool** which contains an entry for the respective image file. The **Link type** is **Link to file**.

Manually creating an empty image pool:

You can insert an image pool object below an application node or below the **Global** node of the **Applications tree** by clicking the green plus button and executing the commands **Add other objects > Image Pool...**. In the **Add Image Pool** dialog box, define a **Name** for the pool.

Adding an image file to an image pool

Adding an image file to an image pool	Actions to be performed
By executing the command **Insert Image**	1. Put the focus into the image pool editor. 2. Execute the **Insert Image** command (see Menu Commands, Online Help) from the contextual menu. **Result**: A unique ID is entered automatically, which is editable. 3. Double-click the field **File name** in the new line to specify the path of the image file. 4. For this purpose, you can open the dialog box **Select Image** by clicking the  button. The edit fields and options of this dialog box are explained below this table. **NOTE:** If you do not use the dialog box **Select Image**, but enter the image file path directly, then automatically the link type setting **Remember the link** is used.
By directly entering the file name	In the editor of the image pool, double-click the field **File name** of the first empty line. Enter - as described above for the first option (executing the command **Insert Image**) - the path of the desired image file. **Result**: The file name is automatically entered as ID.
By drag&drop from the file system	In the local file system browser, select the desired image file and drag it into the image pool editor. Multiple selection is possible. **Result**: The file name is automatically entered as ID. The link type setting **Remember the link** is used automatically.

Elements of the **Select Image** dialog box:

Element	Description
Image File	Enter the path of the image file or click the  button for getting the standard dialog box for browsing the local file system. Select the desired file or files. Multiple selection is possible.
File Handling	Choose a link type: • **Remember the link**: The file is only available in the project if it is available in the specified path. Files specified without path must be stored in the project folder. • **Remember the link and embed into project**: A copy of the file is stored internally in the project. The link to the specified path is stored as well. As long as the image file is available under the stored path, the update action as defined below, is valid. As soon as the image file is removed from the specified location, only the copy of the file stored internally in the project will be used. • **Embed into project**: Only a copy of the file is stored internally in the project. The link to the external path is not stored. If you choose the option **Remember the link and embed into project**, you can select one of the following update actions in the **Change Tracking** section: • **Reload the file automatically.** • **Prompt whether to reload the file.** • **Do nothing.**

Using Images Which Are Managed in Image Pools

If the ID of the image to be used is specified in multiple image pools:

- search order: If you choose an image managed in the **GlobalImagePool**, you do not need to specify the pool name. The search order for images corresponds to that for global variables:

  1. **GlobalImagePool**
  2. image pools assigned to the currently active application
  3. image pools in **Global** node of the **Applications tree** besides **GlobalImagePool**
  4. image pools in libraries

- unique accessing: You can directly call the desired image by adding the image pool name before the ID according to syntax: <pool name>.<image ID> (For an example, see `imagepool1.drive_icon` in the previous graphic.)

Using an Image in a Visualization Element of Type Image

When inserting an image element in a visualization, you can define it to be a static image or a dynamic image. The dynamic image can be changed in online mode according to the value of a project variable:

Static images:

In the configuration of the element (property **Static ID**), enter the image ID or the image pool name + image ID. Consider in this context the remarks on search order and unique accessing in the previous paragraph.

Dynamic images:

In the configuration of the element (property **Bitmap ID variable**), enter the variable which defines the ID, for example, `PLC_PRG.imagevar`.

Using an Image for the Visualization Background

In the background definition of a visualization, you can define an image to be displayed as visualization background. The image file can be specified as described previously for a visualization element by the name of the image pool and the image file name.

Application

Application

Overview

An application is a set of objects which are needed for running a particular instance of the controller program on a certain hardware device (controller). For this purpose, independent objects managed in the **Global** node of the **Applications tree** are instantiated and assigned to a device. This meets the concept of object-orientated programming. However, you can also use purely application-specific POUs.

An application is represented by an application object in the **Applications tree**. Below an application entry, insert the objects defining the application resource set.

One application is available for each controller. It is not possible to add further applications.

A part of each application is the **Task Configuration** controlling the run of a program (POU instances or application-specific POUs). Additionally, it can have assigned resource objects like global variable lists, libraries, and so on. These - in contrast to those managed in the **Global** node of the **Applications tree** - can only be used by the particular application. For the rules, refer to the description of arranging and configuring objects in the **Devices tree**, page 40.

Consideration

When going to log in with an application on a target device (controller or simulation target), two checks are performed: Which application is currently in the controller? Are the application parameters in the controller matching those in the application within EcoStruxure Machine Expert? Corresponding messages indicate mismatches and offer some ways to continue in this case. Also you have the possibility to delete the application in the controller. Refer to the description of the *Login* command, page 237 for more details.

Task Configuration

What's in This Chapter

Task Configuration 233
Adding Tasks 233

Task Configuration

Overview

The **Task Configuration** defines 1 or several tasks for controlling the processing of an application program.

It is a resource object for an application, page 232. It has to be inserted in the **Applications tree** below an application node. A task can call an application-specific program POU, which is only available in the **Applications tree** below the application. It can also call a program which is managed in the **Global** node of the **Applications tree**. In the latter case, the program that is available globally will be instantiated by the application.

You can edit a task configuration in the **Task Configuration** editor, page 404.

In online mode, the **Task Configuration** editor provides a monitoring view that presents information on cycles, cycle times, and task status.

As an additional functionality of the task configuration, if supported by the device, the monitoring view allows a dynamic analysis of the POUs which are controlled by a task. It supplies information about the cycle times, the quantity of function block calls and the unused code lines.

Adding Tasks

Introduction

You can add tasks to your application via the **Applications tree**.

Procedure

Step	Action
1	In the **Applications tree**, select the **Task Configuration** node, click the green plus button, and execute the command **Task...**. Alternatively, you can right-click the **Task Configuration** node, and select **Add Object > Task...** from the contextual menu. **Result:** The **Add Task** dialog box opens.
2	In the **Add Task** dialog box, enter a name in the **Name:** text box. **Note:** The name must neither contain any space nor exceed a length of 32 characters.
3	Click **Add**.

Managing Applications

What's in This Chapter

General Information 234
Building and Downloading Applications 235
Running Applications 250
Maintaining Applications 251

General Information

Introduction

Introduction

To run an application, you must first connect the PC to the controller, then download the application to the controller.

NOTE: Due to memory size limitation, some controllers are not able to store the application source but only a built application that is executed. Therefore, you are not able to upload the application source from the controller to a PC.

⚠WARNING

UNINTENDED EQUIPMENT OPERATION

- Confirm that you have entered the correct device designation or device address in the **Communication Settings** dialog when downloading an application.
- Confirm that machine guards and tags are in place such that any potential unintended machine operation will not result in personal injury or equipment damage.
- Read and understand all user documentation of the software and related devices, as well as the documentation concerning equipment or machine operation.

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

Preconditions

Verify that your application meets the following conditions before downloading it to the controller:

- the active path is set for the correct controller,
- the application you want to download is active,
- the application is free of compilation errors,
- the **Online > Operating Mode** is set to **Debug**. For further information, refer to the **Operating Modes** chapter in the Menu Commands Online Help.

Boot Application

The boot application is the application that is launched on controller start. This application is stored in the controller memory. To configure the download of the boot application, right-click the **Application** node in the **Applications tree** and select the **Properties** command.

At the end of a successful download of a new application, a message is displayed asking you if you want to create the boot application.

You can manually create a boot application in the following ways:

- In offline mode: Click **Online > Create boot application** to save the boot application to a file.
- In online mode, with the controller being in STOP mode: Execute the **Online > Create boot application** command (see Menu Commands, Online Help) to download the boot application to the controller.

Building and Downloading Applications

Building Applications

Overview

You can perform an offline code generation in order to verify whether compilation errors are detected before downloading the code to the device. For a successful login, the code generation must have been completed without detecting errors.

Code Generation, Compile Information

Machine code is generated in the following cases:

- When the Application, page 232 project is downloaded to the target device (controller, simulation target).
- When the **Build > Generate code** command is executed.

At each download, the compile information, containing the code and a reference ID of the loaded application, is stored in the project directory in a file *<projectname>.<devicename>.<application ID>.compileinfo*. The *compileinfo* file is deleted when the **Clean** or **Clean all** command is executed.

The build process verifies the project in order to detect programming errors. Detected programming errors are displayed in the **Messages** view (message category **Build**).

Messages at Code Generation

At each code generation, information about memory allocation is displayed in the **Messages > Build** view.

Memory allocation information		Description
Memory area <n>		The contents of the reserved memory areas are displayed: The memory area in which the various types of data and the code are stored depends on the controller. **NOTE:** The code and data area consumption is contained in a message similar to: `[INFORMATION] Memory area 4 contains Data, Code and Nonsafe data: size: 10436608 bytes, highest used address: 107752, largest contiguous memory gap: 10328856 bytes (98 %)` For the addresses `%I`, `%M`, `%Q`, memory is allocated, even if no variable is assigned to an address. In the **Messages** view, the memory areas for `%M`, `%I`, `%Q` are indicated as **Memory**, **Input** and **Output**. When you run a **Build > Clean** command, the memory is reallocated and defragmented. Memory fragmentation (gaps in the memory) is caused by consecutive incremental builds and online changes. This leads to the following effects: • The number of memory gaps increases. • The largest contiguous memory gap decreases.
	Highest used address (Byte)	The highest reserved address in the allocated memory area is displayed. During the first build after a cleaning operation, the memory addresses are assigned to variables in ascending order, taking into consideration the natural alignment of the controller (usually 8 bytes). As a result, the highest address used at this time corresponds approximately to the amount of memory used. The rest of the allocated memory area is at this time available for incremental builds and online changes.
	Largest contiguous memory gap (in bytes)	The memory size that is effectively available is displayed in bytes and as percentage of the total memory. Gaps that were created in the allocated memory are reused whenever possible when changes are made. When, for example, a global variable of type Byte is added, it is placed in the first free byte of the memory. Other elements, such as a function block instance, a variable of the type structure or array, or the code for a POU, need to be stored contiguously and occupy more memory. Thus, they cannot fill such small gaps as the variable of type Byte but must be allocated to the largest contiguous memory gap area. **NOTE:** Also refer to the **Online Change Reserve Memory Settings** command, page 195 for how to avoid memory re-allocation of function block instances in case of small modifications on a function block type.

Login

Overview

The **Online > Login** command connects the application to the target device (controller or simulation target) and thus changes into the online mode.

The default shortcut is **Alt** + **F8**.

⚠WARNING

UNINTENDED EQUIPMENT OPERATION

- Confirm that you have entered the correct device designation or device address in the “Communication Settings” dialog when downloading an application.
- Confirm that machine guards and tags are in place such that any potential unintended machine operation will not result in personal injury or equipment damage.
- Read and understand all user documentation of the software and related devices, as well as the documentation concerning equipment or machine operation.

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

If there is an online user management (also refer to the chapter *Users and Groups*, page 132) established on the target device, at login you are prompted to enter the user name and password. For this purpose, the dialog box **Device User Logon** opens.

NOTE: If a password has expired, you are prompted to configure a new password.

NOTE: After several attempts to log in with incorrect credentials, your account may get locked for a time configured in the **Change Runtime Password Policy** dialog box, page 116.

NOTE: The layout of views can change when you connect to the target device because the **Online** perspective, page 49 is automatically selected when you switch to online mode.

NOTE: If the communication with the controller is encrypted or secured using user management functions, refer to the *How to Manage Certificates on the Controller User Guide* for further information.

Selecting the View of the Communication Settings Tab in the Device Editor

Depending on the option selected for **Communication page** in the **Tools > Options > Device editor** dialog box (see Menu Commands, Online Help), the **Communication Settings** tab of the device editor is displayed in three different modes. Refer to the description of the different tabs for details:

Mode	Communication page setting	Dialog of the device editor
1	**Controller selection mode** (default setting)	**Communication Settings** tab in controller selection mode, page 87
2	**Simple mode**	**Communication Settings** tab in simple mode, page 111
3	**Classic mode**	**Communication Settings** tab in classic mode, page 117

Login Procedure

The steps of the login procedure are identical. They only differ in the presentation of the **Communication Settings** tab.

For a successful login, the code generation must have been completed without detecting errors (refer to the chapter *Build Process Before Login*, page 238).

Step	Action
1	Execute the command **Online > Login**, or click the **Login** button  from the toolbar, or press ALT + F8. **Result**: Since no target address has been set before, the **Communication Settings** tab of the device editor opens in the mode you selected. A message box is displayed indicating that a valid address has not been defined.
2	If only one controller has been detected, this controller is used as target device. If several controllers have been detected, double-click the controller you want to log in. **NOTE:** In controller selection mode and in classic mode, only those controllers are listed that have the same **Target ID** as the selected controller. To display all controllers in the list, set the **Filter** criterion to **None**.
3	Execute the command **Online > Login**, or click the **Login** button  from the toolbar, or press ALT + F8. **Result**: A message box displays to inform you of potential hazards.
4	Click **Cancel** to abort the login operation or press ALT + F to confirm the message and to log in to the selected controller. **Result**: If you press ALT + F the connection to the controller is established, and you can download the application, page 239.

Unsuccessful Login

If...	Then ...	Comment
If an error is detected during login to the controller,	Then the operation is aborted and an error message is displayed.	In the error message, you can display the error details.
If an exception has occurred containing the text **SOURCEPOSITION** in the log message,	Then you can execute the command **Show source code in editor**.	This command opens the function in the editor with the cursor at the position where the error was detected.

Build Process at Changed Applications

Build Process Before Login

Before **Login** and if the current affected application project has not been compiled since having been opened or since the last modification, it will be compiled. Therefore, the project will be built correspondingly to a **Build** run in offline mode and compilation code for the controller will be generated.

If errors are detected during compilation, a message box opens with the following text: **There are compile errors. Do you want to login without download?** You can choose to correct the detected errors first, or to login nevertheless. In the latter case, you are logged in to that version of the application which is possibly already available on the controller.

The detected errors are listed in the **Messages** view (category **Build**).

Downloading an Application

Introduction

To run an application, first connect the PC to the controller, then download the application to the controller.

Downloading a project allows you to copy the current project from EcoStruxure Machine Expert to the controller memory.

NOTE: Due to memory size limitation, some controllers are not able to store the application source but only a built application that is executed. Therefore, you are not able to upload the application source from the controller to a PC.

⚠WARNING

UNINTENDED EQUIPMENT OPERATION

- Confirm that you have entered the correct device designation or device address in the Communication Settings dialog when downloading an application.
- Confirm that machine guards and tags are in place such that any potential unintended machine operation will not result in personal injury or equipment damage.
- Read and understand all user documentation of the software and related devices, as well as the documentation concerning equipment or machine operation.

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

Preconditions

Verify that your application meets the following conditions before downloading it to the controller:

- The active path is set for the correct controller.
- The application you want to download is active.
- The application is free of compilation errors.
- The **Online > Operating Mode** is set to **Debug**. For further information, refer to the **Operating Modes** chapter in the Menu Commands Online Help.

Boot Application

The boot application is the application that is launched on controller start. This application is stored in the controller memory. To configure the download of the boot application, right-click the **Application** node in the **Devices** view and select the **Properties** command.

At the end of a successful download of a new application, a message is displayed asking you if you want to create the boot application.

You can manually create a boot application in the following ways:

- In offline mode: Click **Online > Create boot application** to save the boot application to a file.
- In online mode, with the application being in STOP mode: Click **Online > Create boot application** to download the boot application to the controller.

Operating Modes

The download method differs depending on the relationship between the loaded application and the application you want to download. The 3 cases are:

- Case 1: The application in the controller is the same as the one you want to load. In this case, no download occurs, you just connect EcoStruxure Machine Expert to the controller.
- Case 2: Modifications have been made to the application that is loaded in the controller in comparison to the application in EcoStruxure Machine Expert. In this case, you can specify if you want to download all or parts of the modified application or keep the application in the controller as it is.
- Case 3: A different or a new version of application is already available on the controller. In this case, you are asked whether this application should be replaced.
- Case 4: The application is not yet available on the controller. In this case, you are asked to confirm the download.

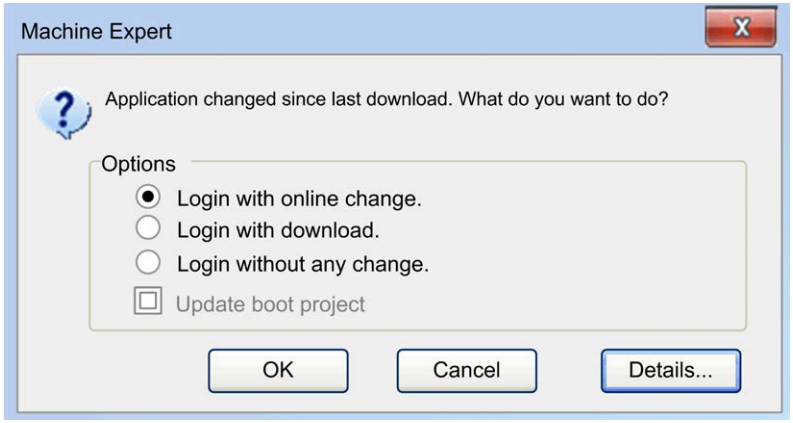
Downloading Your Application to the Controller: Case 1

The application in the controller is the same as the one you want to load. In this case, no download occurs, you just connect EcoStruxure Machine Expert to the controller.

Step	Action
1	To connect to the controller, select **Online > Login to 'Application[YourApplicationName; Plc Logic]'**.
2	You are connected to the controller.

Downloading Your Application to the Controller: Case 2

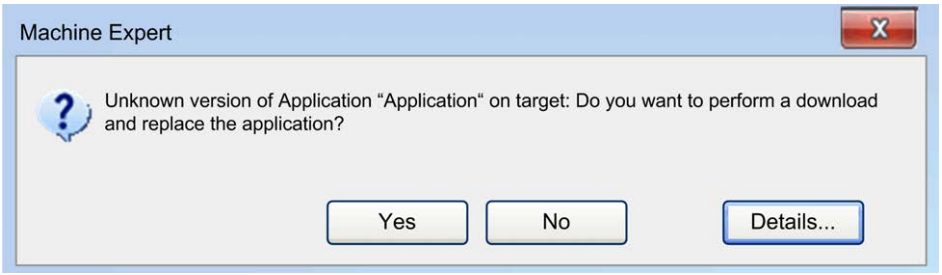
Modifications have been made to the application that is loaded in the controller in comparison to the application in EcoStruxure Machine Expert.

Step	Action
1	To connect to the controller, select **Online > Login to 'Application[YourApplicationName; Plc Logic]'**.
2	In case you modified your application, and you want to reload it into the controller, the following message appears:  **Login with online change** Only the modified parts of an already running project is reloaded to the controller. **Login with download** The whole modified application is reloaded to the controller. **Login without any change** The modifications are not loaded. **NOTE:** If you select the option **Login without any change**, the changes you perform in the EcoStruxure Machine Expert application are not downloaded to the controller. In this case, the information and status bar will show **RUNNING** as operational state and will indicate **Program modified (Online change)**. This differs from the options **Login with online change** or **Login with download**, where the information and status bar indicates **Program unchanged**. In this case, monitoring of variables is possible, but the logic flow may be confusing because the values on function block outputs may not match to the values on the inputs. **Examples** In LD, contact states are monitored based on the affected variables. This may have the effect that a blue animated contact followed by a blue link (meaning true) is shown, although the coil connected to this contact shows it as false.In ST logic flow, an IF statement or a loop seems to be executed, but it is actually not executed because the condition expression is different in the project and on the controller.
3	Select the suitable option and click **OK**.

NOTE: Consult the *Programming Guide* specific to your controller for important safety-related information concerning the downloading of applications.

Downloading Your Application to the Controller: Case 3

A different or a new version of application is already available on the controller.

Step	Action
1	To connect to the controller, select **Online > Login to 'Application[YourApplicationName; Plc Logic]'**.
2 a	In case, the controller is not in RUN mode, and you want to load a different application than the one currently in the controller, the following message appears:  Refer to the hazard messages below before you click **Yes** to download the new application to the controller, or **No** to cancel the operation.
2b	In case, the controller is in RUN mode, and you want to load a different application than the one currently in the controller, the following message appears:  Refer to the build messages below before you click **Yes** to download the new application to the controller, or **No** to cancel the operation.

⚠WARNING

UNINTENDED EQUIPMENT OPERATION

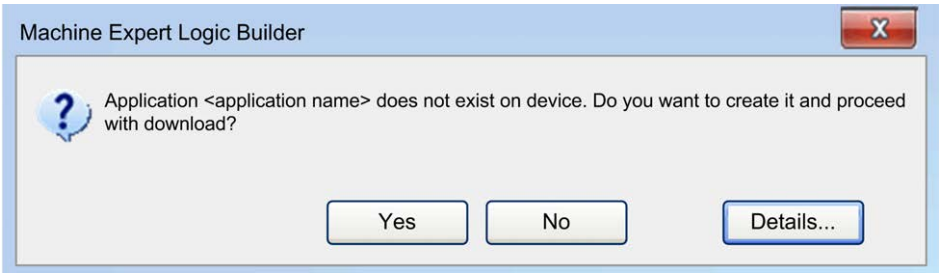
Verify that you have the correct application before confirming the download.

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

If you click **Yes**, the application running in your controller will be overwritten.

Downloading Your Application to the Controller: Case 4

The application is not yet available on the controller.

Step	Action
1	To connect to the controller, select **Online > Login to 'Application[YourApplicationName; Plc Logic]'**.
2	In case the application is not yet available on the controller, you are asked to confirm the download. For this purpose, a dialog box with the following text displays:  Click **Yes** to download the application to the controller, or **No** to cancel the operation.

NOTE: Consult the *Programming Guide* specific to your controller for important safety-related information concerning the downloading of applications.

Online Change

The **Online Change** command modifies the running application program and does not affect a restart process:

- The program code can behave other than after a complete initialization because the machine keeps its state.
- Pointer variables keep their values from the last cycle. If there is a pointer on a variable, which has changed its size due to an online change, the value will not be correct any longer. Verify that pointer variables are reassigned in each cycle.

⚠WARNING

UNINTENDED EQUIPMENT OPERATION

Thoroughly test your application code for proper operation before placing your system into service.

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

NOTE: Consult the *Programming Guide* specific to your controller, chapter ***Controller States Description*** for specific information.

If the application project currently running on the controller has been changed in the programming system since it has been downloaded last, just the modified objects of the project will be loaded to the controller while the program keeps running.

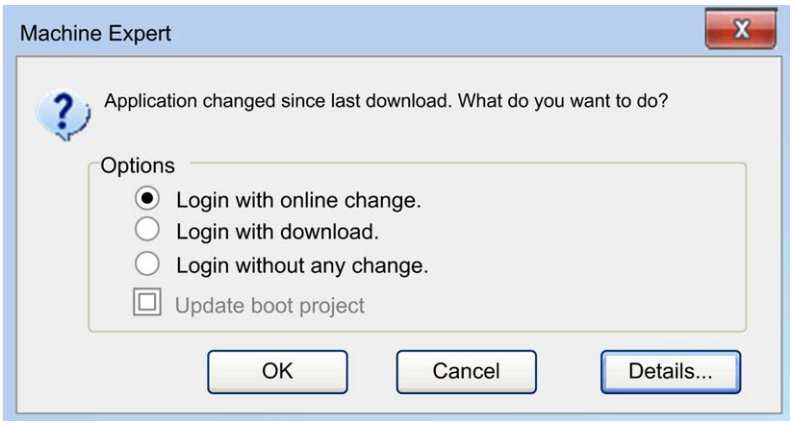
In the **Online Change Memory Reserve** view, page 195, you configure a memory reserve for the online change of function blocks. After you have made modifications on a function block and you perform an online change, it is no longer necessary to copy the instance variables of the function block to a new memory area.

NOTE: With EcoStruxure Machine Expert V2.2 and later versions, only one instance is allowed to log in to an application of a controller. An error message is displayed when a login attempt is made by a second instance.

Implicit Online Change

When you try to log in again with a modified application (checked via the COMPILEINFO, which has been stored in the project folder during the last download), you are asked whether you want to make an online change, a download, or login without changing.

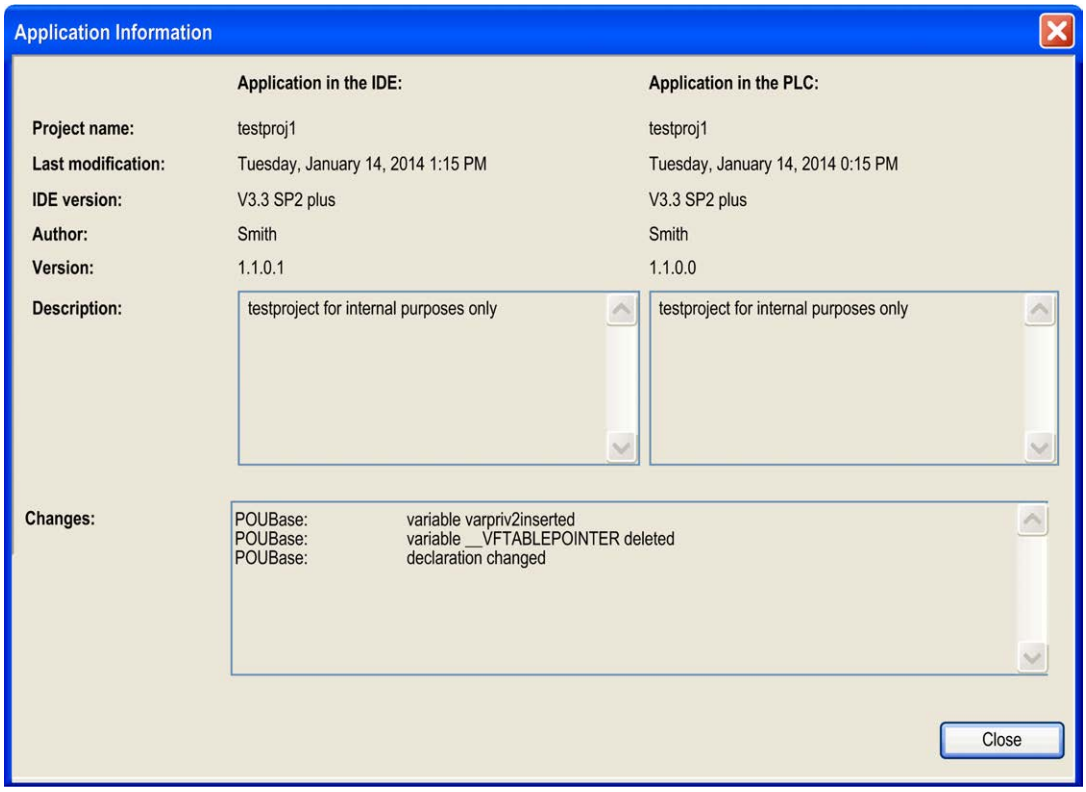
Login dialog box:



Description of the elements:

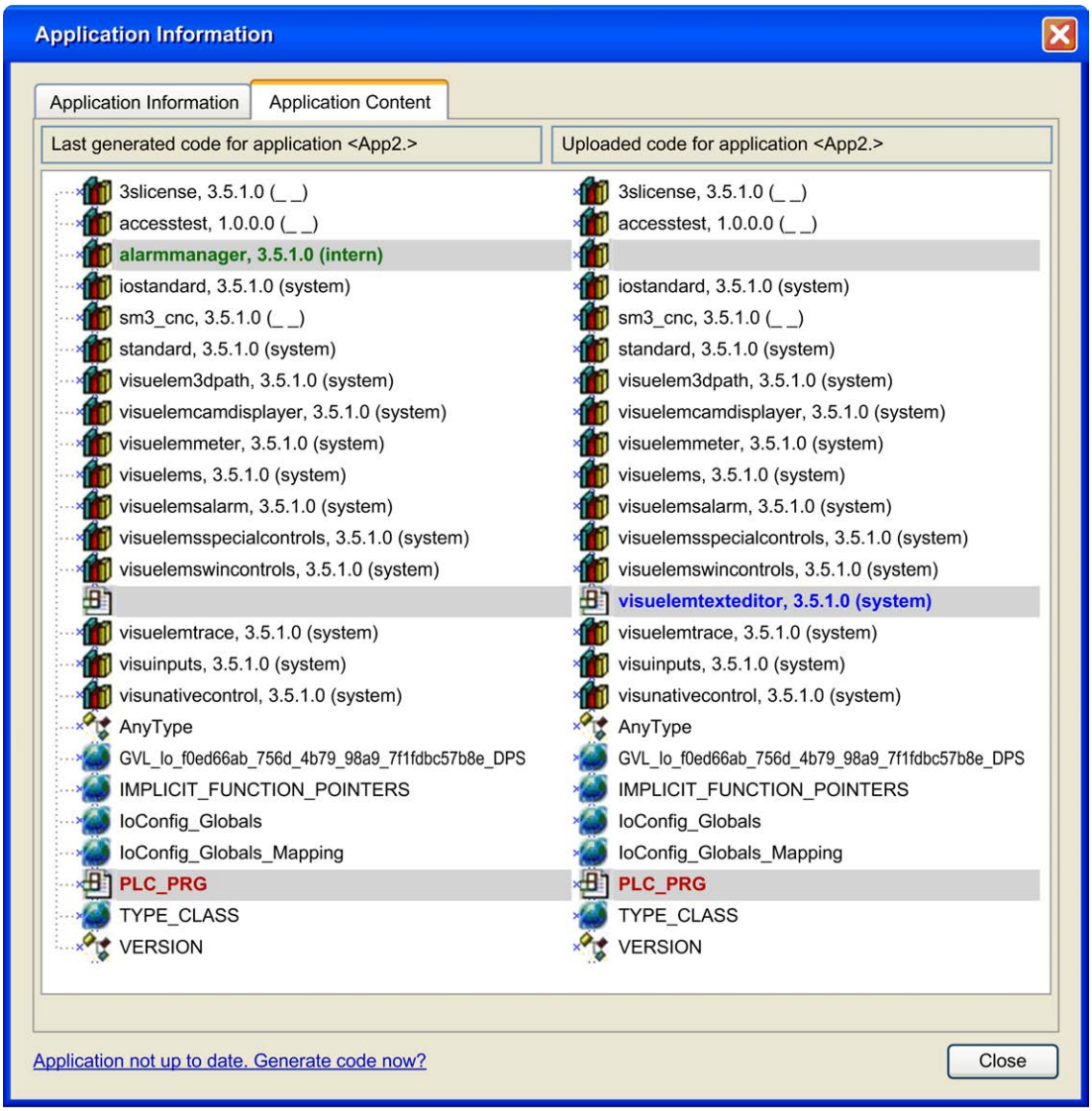
Element	Description
Login with online change	This option is selected per default. If you confirm the dialog box by clicking **OK**, the modifications will be loaded and immediately displayed in the online view (monitoring) of the respective object or objects.
Login with download	Activate this option to load and initialize the application project completely.
Login without any change	Activate this option in order to keep the program running on the controller unchanged. Afterwards, an explicit download can be performed, thus loading the complete application project. It is also possible that you are asked again whether an online change should be performed at the next relogin.
Update boot project	This option is by default not selected. To select this option, activate the option **Implicit boot application on Online Change** in the **Boot Application** tab of the **Properties** dialog box of the **Application** node. A boot application is then automatically created with an online change.
Details	Click this button to obtain the **Application Information** dialog box (**Project name, Last modification, IDE version, Author, Description**) on the current application within the IDE (Integrated Development Environment, i.e., EcoStruxure Machine Expert) in comparison to that currently available on the controller. Refer to the following figure.

Application Information dialog box



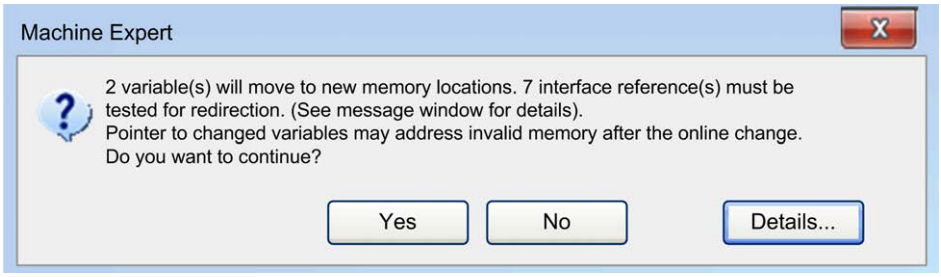
For further information, refer to the *Login* chapter, page 237.

Application Content tab of the **Application Information** dialog box:



If the option **Download Application Info** is activated in the **Application Build Options** tab of the **View > Properties** dialog box (see Menu Commands, Online Help), this tab shows the following: The content of the application read from the controller is shown in the column **Uploaded code for application <App2>** and can be compared to the content of the application in the programming system. To update the left column **Last generated code for application <App2>** with the latest version of the application active in the programming system, click the button **Application not up to date. Generate code now?**. The contents of the applications are compared and different objects are marked with colors as they are in the **Project > Compare** function (see Menu Commands, Online Help). This more granular information can help you to evaluate the effects downloading the new application.

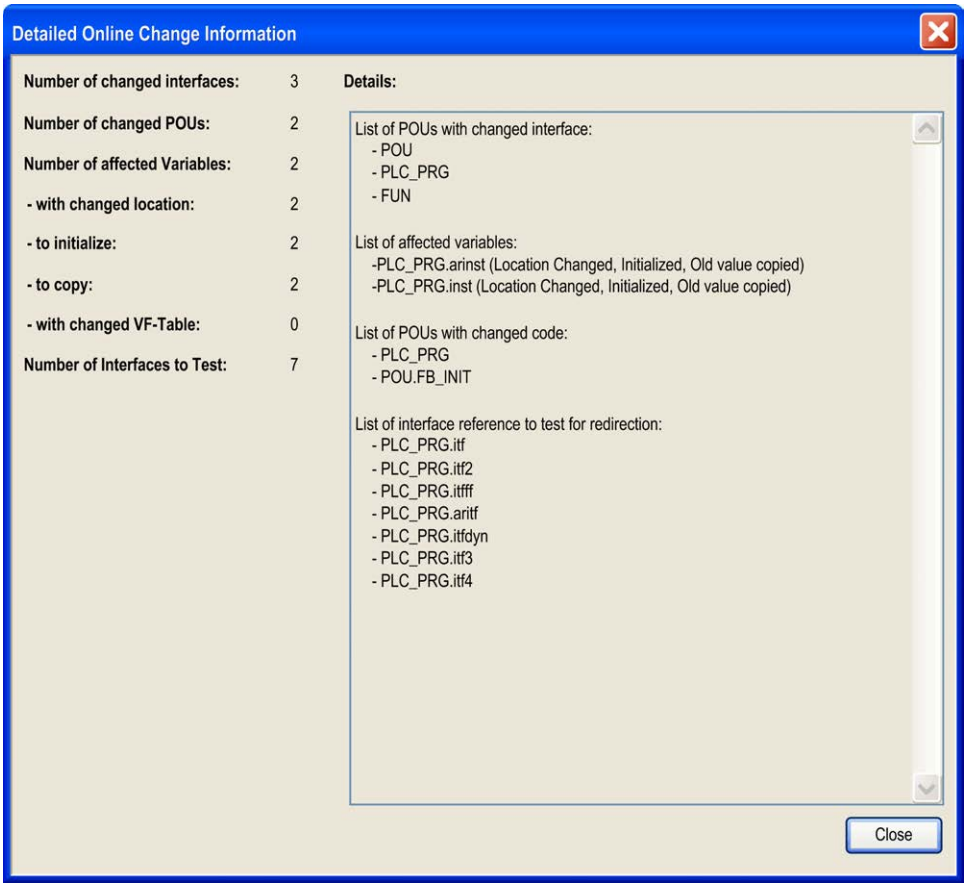
If the online change will affect considerable changes in download code, like for example possible moves of pointer addresses or necessary redirections of interface references, page 186 another message box is displayed after you have confirmed the **Online change** dialog box with **OK** before download will be performed. It informs you about the effects you have to consider and provides the option to abort the online change operation.



NOTE: After having removed implicit check function (such as CheckBounds) from your application, no **Online Change** is possible, just a download. A corresponding message will appear.

Click the **Details** button in this message box to display detailed information, such as the number and a listing of changed interfaces, POUs, affected variables, and so on.

Detailed Online Change Information dialog box



Explicit Online Change

Execute the command **Online Change** (by default in the **Online** menu) to explicitly perform an online change operation on a particular application.

An **Online Change** of a modified project is no longer possible after a **Clean** operation (**Build > Clean all**, **Build > Clean**). In this case, the information on which objects have been changed since the last download will be deleted. Therefore, only the complete project can be downloaded.

NOTE: Consider the following before executing the **Online Change** command:

- Verify that the changed code is free from logical errors.
- Pointer variables keep their value from the last cycle. If you point to a variable which now has been deplaced, the value will no longer be correct. For this reason, reassign pointer variables in each cycle.

Information on the Download Process

When the project is loaded to the controller completely at **Login** or partially at **Online Change**, then the **Messages** view will show information on the generated code size, the size of global data, the needed memory space on the controller and in case of online change also on the affected POUs.

NOTE: In online mode, it is not possible to modify the settings of devices or modules. To change parameters of the devices, the application must be logged out. Depending on the bus system, there can be some special parameters which are allowed to be changed in online mode.

Boot Application (Boot Project)

At each successful download, the active application is automatically stored in a file <*application name*>.*app* in the controller system folder, thus making it available as a boot application. The boot application is started automatically when the controller is started (booted). To make the download of the active application the boot application, you must execute the command **Create boot application** (available in the **Online** menu).

You can also create the boot application while in offline mode, page 239.

If you want to connect to the same controller from the programming system on different PC, or, retrieve the active application from a different PC, without the need of an online change or download, follow the steps described in the *Transferring Projects to Other Systems* paragraph.

Transferring Projects to Other Systems

For transferring a project to another computer, use a project archive (see Menu Commands, Online Help).

You can transfer a project, which is already running on a controller *xy*, from the programming system on PC1 to that on PC2. To be able to reconnect from PC2 to the same controller *xy* without the need of an online change or download, verify the following project settings before creating a project archive.

Perform the following steps:

1. Verify that only libraries with definitive versions are included in the project, except for the pure interface libraries. (Open the **Library Manager** and check entries with an asterisk (*) instead of a fix version (see Functions and Libraries User Guide).)
2. Ensure that a definitive compiler version is set in the **Project Settings > Compile options** dialog box (see Menu Commands, Online Help).
3. Make sure that a definite visualization profile is set in the **Project Settings > Visualization Profile** dialog box (for more information, refer to the *Visualization* part of the online help).
4. Verify that the application currently opened is the same as that already available on the controller. That is, the boot project (refer to the **Online > Create boot application** command (see Menu Commands, Online Help)) must be identical to the project in the programming system. If there is an asterisk behind the project title in the title bar of the programming system window, the project has been modified but not yet saved. In this case, it can differ from the boot project. If necessary, before transferring the project to another PC, create a (new) bootproject - for some controllers this is done automatically at a download - and then download and start the project on the controller.
5. Create the project archive with the following information: **Download information files**, **Referenced devices**, **Referenced libraries**, **Visualization Profile**.
6. Log out. If necessary, stop and restart controller *xy* before reconnecting from PC2.
7. Extract the project archive on PC2 with the same information options activated as listed in step 5.

   NOTE: For login without online change, make sure that you use the EcoStruxure Machine Expert version that was used to create and download the application in the controller.

For further information, refer to the chapter *Project Archives Helping to Preserve Compatibility* in the Compatibility and Migration User Guide for EcoStruxure Machine Expert.

Managing Expired Certificates

Overview

Certificates have a limited validity and therefore need to be updated at regular intervals.

If the certificate of the controller for encrypted communication has expired (refer to the validity information **not before** until **not after**), a message is displayed when you attempt to access the controller. To login anyway, you can accept the expired certificate and connect to the controller. The message will be displayed every time you attempt to login until you install a valid certificate.

If a new certificate is available on the controller (generated or imported), it will be presented during next login for you to accept.

NOTE: Encrypted communication with other clients (for example, PLCHandler) will not be established as they typically do not accept an expired certificate.

Renewing a Certificate (While the Present Certificate is Still Valid)

It is a good practice to provide a new certificate before the present certificate expires to help provide seamless encrypted communication.

As soon as you attempt to login to a controller where a new certificate is available in parallel to the certificate that is being used, you are prompted to accept this new certificate. After you have accepted the new certificate, it is being used for encrypted communication with this controller.

Running Applications

Running Applications

Introduction

This part defines how to start/stop an application.

RUN/STOP with EcoStruxure Machine Expert

The controller can be started and stopped using EcoStruxure Machine Expert run on a PC connected to the controller.

Click **Online > Start ‘Application [ApplicationName: Plc logic]’** or **F5** key or the **Start ’Application [ApplicationName: Plc logic]’** button in the menu bar to start the application.

Click **Online > Stop ‘Application [ApplicationName: Plc logic]’** or **Shift** + **F8** keys or the **Stop ’Application [ApplicationName: Plc logic]’** button in the menu bar to stop the application.

RUN/STOP Input for Controllers

This function is not available for all supported controllers. Consult the *Programming Guide* specific to your controller (for example, Run/Stop chapter (see Modicon M251 Logic Controller, Hardware Guide)).

RUN/STOP Switch for Controllers

This function is not available for all supported controllers. Consult the *Programming Guide* specific to your controller (for example, Run/Stop chapter (see Modicon M251 Logic Controller, Hardware Guide)).

Maintaining Applications

Monitoring

Overview

In online mode, there are various possibilities to display the current values of the objects in the controller:

- You can see the values of the objects in a program editor screen while online. For details, see the description of the respective editor.
- You can view object values in the online view of the declaration editor. For details, refer to the description of the declaration editor, page 373.
- You can view objects independently in lists through the command **Watch**. For details, refer to the description of the watch view / watch list editor, page 423. You can insert a variable in a watch view by selecting it and executing the command **Add watchlist** from the contextual menu.
- You can view values via trace sampling: recording and displaying of variable values from the controller. For details, refer to the description of the trace object functionality, page 454.
- You can view object values that are contained in recipes: User-defined set of variables for writing and watching these variables on the controller. Refer to the description of the recipe manager, page 434.

For information on monitoring of properties that are inserted beneath POUs or function blocks, refer to the chapter Property, page 184.

For information on monitoring of function calls, refer to the chapter *Attribute Monitoring*, page 615.

NOTE: If a value is not valid (for example, the result of calculating the square root of a negative number), the result may be displayed as `NaN` (not a number) or `INF` (infinite value) depending on the operation, the object, and the particular controller platform. Consult the *Programming Guide* specific to your controller for more information.

Debugging

Overview

To evaluate potential programming errors, you can use the debugging functionality.

For testing purposes, an application can also run in simulation (see Menu Commands, Online Help) without any required link to an actual target device. While simulation avoids the need to connect to physical hardware, there are limitations to which you may need to complete debugging online.

NOTE: If you are using debugging with breakpoints in a bus cycle task for the following protocols, disable the timeout control (watchdog) in the Watchdog Settings, page 415.

- **Profibus Master**
- **Profinet Controller**
- **EthernetIP Scanner**

NOTE: Ensure to re-enable the timeout control (watchdog) in the Watchdog Settings, page 415 after debugging is complete.

Breakpoints for Forcing Execution Stops

A breakpoint that is set in an application program stops program execution. The variable values can then be seen at the current execution point. The current position can be determined more precisely using the Call Stack (see Menu Commands, Online Help).

Only the debug task that has just reached the breakpoint is stopped. The other tasks continue to execute. The possible breakpoint positions depend on the respective program editor. In each case, there is a breakpoint at the end of a POU.

> **NOTE:** The inputs / outputs handled by the debug task are not updated when stopped at a breakpoint. This applies even if the option **Update IO while in stop** is enabled in the **PLC settings [GENERAL]** tab of the device editor, page 128.

Refer to the chapter *Breakpoints Commands* for a description of the commands concerning breakpoints. The **Breakpoints** dialog box (see Menu Commands, Online Help) provides an overview on all breakpoints, allowing you to add, remove, and modify breakpoints.

Conditional Breakpoints

The stop at the breakpoint can be dependent on the following conditions:

- At the time, a specific boolean expression is TRUE.
- Which cycle number is running.
- Which task is executing.
- When the value of a variable is modified (data breakpoint).

Declaring a specific debug task can help to avoid that several tasks are affected by the error check (refer to the *Breakpoints and Stepping in Applications with Multiple Tasks* paragraph, page 255).

Execution Points

In online mode, the **Breakpoints** dialog box (see Menu Commands, Online Help) allows you to change a breakpoint and data breakpoint in such a way that it becomes an execution point. Then the program does not stop at the position, but the code processing is initiated.

Symbols

Symbol	Description
	Breakpoint enabled.
	Breakpoint disabled.
	The breakpoint is set in another instance of the function block that is open in the editor.
	Stop at breakpoint.
	Conditional breakpoint enabled.
	Conditional breakpoint disabled.
	Execution point enabled.

Symbol	Description
	Execution point disabled.
	Conditional execution point enabled.
	Conditional execution point disabled.
	Current step position. Indicated by a yellow arrow before the respective line and a yellow shadow behind the concerned operation.
	Data breakpoint enabled.
	Data breakpoint disabled.
	Stop at data breakpoint.
	Data execution point enabled.
	Data execution point disabled.
	Stop at data execution point.
	Conditional data execution point enabled.
	Conditional data breakpoint enabled.

Stepping Through the Program

Stepping allows a controlled execution of an application program in online mode, for debugging purposes. Before you can use stepping commands, the program must be stopped at a defined program step, for example by setting a breakpoint. Basically, you step from one instruction to the next by step into an instruction, step over the next instruction or step out of an instruction. Refer to the chapter *Breakpoint-Related Commands* (see Menu Commands, Online Help) for a description of the stepping commands.

The following stepping functions are provided:

- **Step into** command (see Menu Commands, Online Help): Execution of a single step; this also applies to POUs called within a step.
- **Step over** command (see Menu Commands, Online Help): Execution of a single step; POUs are executed completely within a step.
- **Step out** command (see Menu Commands, Online Help): If the application program does not contain any calls, a jump back to the start of the application is executed.
- **Run to Cursor** command (see Menu Commands, Online Help): Execution of the program up to a temporary definable position.
- **Set next statement** command (see Menu Commands, Online Help): Definition of the next instruction (statement) to be executed.
- **Show next statement** command (see Menu Commands, Online Help): Jump to the instruction (statement) to be executed in the next step.
- **Call Stack** view (see Menu Commands, Online Help): This view indicates the currently reached step position with the complete call path.

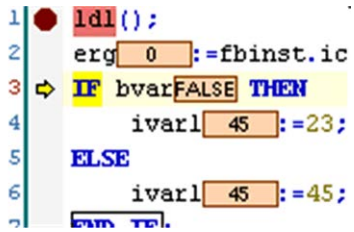
The possible stop positions when stepping through an application program depend on the editor type. The current position is indicated by the yellow arrow.

The **Call Tree** view (see Menu Commands, Online Help) shows the location of the function block in the call structure of an application program even before compiling the application.

Example of a Step Into Operation

Starting from the breakpoint you can execute each single command line with the stepping command.

Step Into, example



Data Breakpoints

Data breakpoints allow you to stop program execution when the value of a variable changes.

This function is not available for all supported controllers. Consult the *Programming Guide* specific to your controller.

The number of data breakpoints you can define depends on the controller you are using. A message is displayed when you attempt to set a data breakpoint and the maximum number has been reached.

The following options exist for data breakpoints as for normal breakpoints:

- Linking the halt at the data breakpoint to an additional condition, refer to *Conditional Breakpoints*, page 252.
- Redefining a data breakpoint to a data execution point where the program does not halt, but processes specific code, refer to *Execution Points*, page 252.

Breakpoints and Stepping in Applications with Multiple Tasks

Debugging cannot be performed on multiple tasks at the same time. While you are working on a task with breakpoints or stepping, breakpoints are ignored in other tasks.

If a breakpoint can be hit by multiple tasks, because the POU is used by multiple tasks, only the task that executes first will be halted. Consider this in case of single stepping or if you continue debugging after a halt. If only one certain task should be concerned (debug task), you can specify it in the breakpoint condition properties (**Breakpoints > New Breakpoint** dialog box, tab **Condition**).

Flow Control for Displaying Exact Values When Processing

You can activate flow control to track parts of the application that have already run. As opposed to standard monitoring, which displays only the variable value between two cycles, the flow control yields the value for each processing step at the moment it is rendered. Also refer to the *Flow Control* description in the *Menu Commands Online Help* (see Menu Commands, Online Help).

Variable Values at Execution Stop

At every execution stop, the values of variables in the current cycle (monitoring) are displayed. To determine the current position, you can display the call stack of a variable. Also refer to the *Call Stack* description in the *Menu Commands Online Help* (see Menu Commands, Online Help).

Core Dump

Core Dump for Exceptions

A core dump is a memory snapshot of the application data. When supported by the runtime system, a core dump is automatically saved to a file *<application_name>.core* in the application directory on the controller when an exception is detected.

In online mode, you can also generate a core dump explicitly if the application is currently stopped at breakpoint or if an exception has been detected. In this case, the core dump file is saved only in the project directory and not on the controller.

In offline mode, you can load the core dump of the application from the controller to the EcoStruxure Machine Expert project. Then the application is displayed in its online view with the data and values at the time when the exception was detected and the core dump was created.

In addition, the log file of the controller can be displayed from the time the core dump was created in the project.

NOTE: The firmware must support the functionality in order to display the variable values in the core dump view correctly. In the online view of the application that is created by loading the core dump into the project, menu commands are displayed as available but do not have any effect in this status. When choosing one of these commands, a message is displayed.

Refer to your controller programming manual to verify whether it supports the core dump functionality.

NOTE: Exception handling in the runtime system may overwrite parts of the variable values in the call stack. As a result, original values may be lost and important information may no longer be displayed when monitoring for variables in functions and methods.

Consider the following, for creating a project archive:

If...	**Then...**
If you created a core dump explicitly by choosing the command **Create Core Dump**,	then an option is automatically available in the archive configuration.
If you copied the core dump from the controller,	then you can add it to a project archive only by choosing the function **Additional files**.

NOTE: A project archive with a core dump must include the download information; otherwise the dump is unusable.

Using the Core Dump to Analyze Exceptions

Prerequisites:

- A project is open with an application that has produced an exception on the controller.
- The runtime system is able to create core dumps.
- The application in the project is in offline mode.

Step	Action
1	Load the core dump from the controller by executing the command **Debug > Load Core Dump**. **Result**: An online view of the application is displayed. You see the variable values and call stack from the time when the error has been detected. This also includes the values of the I/O variables in the mapping dialog of the device configurator and the task configuration. The message **Core dump loaded** is displayed in the status line. The core dump file is copied as *<project_name>.<device_name>.<application_name>.<application-Guid>.core* to the local project directory.
2	Execute the command **Debug > Load device log from Core Dump** to load the device log from the time the error has been detected into the project. **Result**: The log view opens (as in online mode of the device configurator) displaying the events at the time the core dump was created.
3	After you have completed the core dump analysis, execute the command **Debug > Close Core Dump**. **Result**: The core dump view of the application closes and the project returns to its views in normal offline mode.

Manually Creating a Core Dump of the Running Application

Prerequisites:

- An application is in online mode.
- The runtime system supports core dump functionality.

Step	Action
1	Execute the command **Debug > Create Core Dump**. **Result**: The creation of a new core dump starts. A progress bar appears in the status line together with a cancel button. A core dump file is saved to the local project directory as *<project_name>.<device_name>.<application_name>.<application-Guid>.core*.

Programmed Error Handling for Exceptions

As an IEC 61131-3 extension, EcoStruxure Machine Expert support specific operators (__TRY, __CATCH, __FINALLY, __ENDTRY) to catch exceptions. You can program statements that will be executed in case an error is detected instead of stopping the program. For further information, refer to the description of these operators, page 733.

This function is not available for all supported controllers. Consult the *Programming Guide* specific to your controller for further information.

Programming Support

Overview

Settings, dialogs and functionalities are provided that facilitate the input of program code and indicate detected errors at an early stage.

For further information, refer to the descriptions of the following functions:

- **Input Assistant**, page 429
- **Intelli-sense**, page 634
- **Auto Declare...** (see Menu Commands, Online Help)
- **Smart Coding** (see Menu Commands, Online Help)
- **Static Analysis Light**, page 262

Refactoring

Overview

Generally refactoring is the process of restructuring and improving existing software code without changing its behavior.

Refactoring in EcoStruxure Machine Expert provides functionalities for renaming object names and variable names. It allows you to show all usage locations of changed objects and variables and to get renamed all those at once, or only individual ones. You can configure whether and at which locations you should get asked for performing refactoring. Explicitly you can use the command **Refactoring > Rename**.

For further information, also refer to *Options, Refactoring* (see Menu Commands, Online Help).

Command Rename

The command **Refactoring > Rename** is used for a global, project wide renaming of object names, and variable names.

It is available in the contextual menu if you set the cursor on a variable name in the declaration or if you right-click one of the following objects in a navigator:

- Functions
- POUs
- GVLs
- Methods
- Properties
- Devices

The command is also available if you select variables and unit conversions within the editor for unit conversions.

The command is not available in online mode.

Process of Renaming

After you have executed the **Refactoring > Rename** command, a dialog is displayed. It shows the locations where the concerned name is used: In the right part of the window the particular locations, in the left part the navigation tree where the concerned object is marked.

You can select the following commands for each location individually:

- **Reject/accept this change**: Reject/accept single changes in the right window.
- **Reject/accept this object**: Reject/accept all changes in the concerned object.
- **Reject/accept whole project**: Reject/accept all changes in the project.

Accepted changes are displayed with a yellow background, rejected changes with gray background.

Command Add <variable>

The command **Refactoring > Add <variable>** allows you to declare variables in a POU and performs an automatic update at its different points of use.

It is available in the contextual menu or in the **Edit > Refactoring** menu if the cursor is placed in a declaration editor. The **Auto Declare** dialog box, page 561 opens.

After you have confirmed the **Auto Declare** dialog box by clicking **OK**, the **Refactoring** dialog box opens. It consists of the tree structure of the project on the left-hand side. The POUs where the new variable is used are highlighted in the tree. You can double-click a POU to open the detailed view on the right-hand side of the dialog box. In this declaration section and implementation of the POU on the right-hand side, new declarations are highlighted in yellow.

Before you decide which changes you want to accept at which location, select the suitable option from the list above the declaration section and implementation of the POU on the right-hand side:

Option	Description
Add inputs with placeholder text	Select this option to insert the standard placeholder text `_REFACTOR_` whenever the new variable occurs in the implementation code. You can search for this standard text later in order to identify the affected locations.
Add inputs with the following value	Select this option to insert an initialization value whenever the new variable occurs in the implementation code. Enter the initialization value in the text box on the right-hand side of the option **Add inputs with the following value**.

In order to accept or reject changes, right-click the changed locations or execute the commands in the left or right part of the dialog box, as described in the *Process of Renaming* paragraph in this chapter.

Examples for Adding a Variable

Example 1:

By refactoring, the `fun` block receives a new input variable `input3` with the initialization value 1. The change has the following effect:

Before refactoring:
```
fun(a + b, 3, TRUE);
fun(input1:= a + b , input2 :=3 , inputx := TRUE);
```

After refactoring:
```
fun(a + b, 3, input3:=1, TRUE);
fun(input1:= a + b , input2 :=3 , input3:=_REFACTOR_,inputx
:= TRUE);
```

Example 2:

By refactoring, the `fun` block receives a new input variable `input3` with the placeholder text `_REFACTOR_`:

Before refactoring:
```
inst(input1 := a + b, input2 := 3, inputx := TRUE);
fun(a + b, 3, TRUE);
```

After refactoring:
```
inst(input1 := a + b, input2 := 3, input3 := _REFACTOR_,
inputx := TRUE);
fun(a + b, 3, input3:=_REFACTOR_, TRUE);
```

Command Remove <variable>

The command **Refactoring > Remove <variable>** allows you to remove an input or output variable from a POU. It is removed automatically from the POU at its different points of use.

The command is available in the contextual menu or in the **Edit > Refactoring** menu if the cursor is placed in the declaration editor on the identifier of the variable to be removed. A dialog box opens that provides information on the removal. If you confirm the message, the **Refactoring** dialog box opens.

When you accept the changes proposed in the **Refactoring** dialog box, the respective input or output variables are removed automatically from the respective POU at its different points of use.

> **NOTE:** In CFC, only the connection is removed between the removed input or output and the block. The input or output itself remains in the chart.

Example for Removing a Variable in ST

In a POU, refactoring removes the `input4` input variable. It is automatically removed from the respective POU at its different points of use:

Before removal:
```
inst(input1 := a + b, input2 := 3, input4 := 1, input5 :=
TRUE);
fun(a + b, 3, 1, TRUE);
```

After removal:
```
inst(input1 := a + b, input2 := 3, input5 := TRUE);
fun(a + b, 3, TRUE);
```

Command Update Referenced Pins

The command **Refactoring > Update Referenced Pins** only takes effect on the CFC editor and the FBD/LD/IL editor. It corresponds to the commands **Reset Pins** and **Update parameters**.

The command **Refactoring > Update Referenced Pins** updates the input and output pins of the POU, at its different points of use, according to the current POU declaration.

It is available in the contextual menu or in the **Edit > Refactoring** menu if the cursor is placed in the POU name, in the first line of the POU declaration or in the device editor.

Command Reorder Variables

The command **Refactoring > Reorder Variables** allows you to change the order of variable declarations within the scopes *VAR_INPUT*, *VAR_OUTPUT*, or *VAR_IN_OUT*.

Example:

```
VAR_INPUT
  invar2 : INT;
  invar1 : INT;
  in : DUT;
  bvar : BOOL;
  invar3 : INT;
```

Proceed as follows to change the order of declarations:

Step	Action
1	Right-click the declaration part and execute the command **Refactoring > Reorder Variables**. **Result**: The **Reorder** dialog box opens with a list of *VAR_INPUT* variables.
2	Drag the *invar1 : INT;* entry to the position before the *invar2.* entry. **Result**: The *invar1* declaration is at the top position.
3	Click **OK** to close the dialog box. **Result**: The **Refactoring** dialog box opens. The affected elements are marked in yellow.
4	Click **OK** to accept the new order of parameters of the function block. Have in mind that the modified order of the parameters will be reflected also at the call usages of the POUs.

Modifying a Variable Declaration and Applying Refactoring Automatically

Refactoring can help you in renaming variables (using **Auto Declare**) as indicated in the following example.

As a prerequisite for this example, a function block *fb_A* with an input *iA* must be available.

Step	Action	Comment
1	Execute the command **Tools > Options...**.	**Result**: The **Options** dialog box opens.
2	Select the category **Refactoring** on the left-hand side.	–
3	In the **Auto Declare** section, select the options **On adding or removing variables, or on changing the scope** and **On renaming variables**.	–
4	Double-click the function block *fb_A*.	–
5	Select a variable in the declaration of *fb_A*, for example, *iA*.	As an alternative, you can place the cursor before or in the variable.
6	Execute the command **Edit > Auto Declare...**.	As an alternative, use the shortcut **Shift** + **F2**. **Result**: The **Auto Declare** dialog box opens. The settings of the variable *iA* are displayed.
7	Edit the **Name** and change *iA* into *iCounter_A*.	**Result**: The **Refactoring - Renaming from iA to iCounterA** dialog box opens. The POUs where the variable is used are highlighted in the tree structure.
8	Click **OK** to apply the renaming and to close the dialog box.	–

Static Analysis Light

Overview

The static analysis function checks the source code of a project for deviations from certain coding directives before the project is downloaded to the target system. This follows the basic idea of the *lint* analysis tool.

You define the desired set of rules in the **Project Settings > Static Analysis Light** dialog box (see Menu Commands, Online Help).

NOTE: The check is automatically performed at each code generation.

Deviations from the rules are displayed as messages of category **Build** in the **Messages** view. The error numbers are displayed as `SA`<number>.

NOTE: The analysis is only done on the application code in the current project. Libraries are not regarded.

NOTE: For GVL variables: If there are several applications in the project, then only objects below the currently active application are checked. If there is only one application in the project, then also any objects in the POUs pool are checked.

Pragma and Attribute for Static Analysis Light

With the help of pragma instructions, you can exclude parts of the code from the check.

Precondition: The rules must be activated in the **Project Settings**.

NOTE: Rule SA0004: Multiple write access on output, cannot be deactivated via pragma.

Pragma `{analysis ...}`

You can use the pragma `{analysis ...}` to disable particular coding rules for the subsequent code lines. For this purpose, place it in the line above the concerned code (disabling) and in the line below the concerned code (re-enabling); the number(s) of the rule or rules to be disabled must be specified preceded by a minus sign (−) for disabling or by a plus sign (+) for re-enabling. Depending on the rule, the pragma can be used in the declaration part or in the implementation part of a programming object.

Syntax

```
{analysis <sign><rule number>|,<other combinations of signs and rules, comma-separated>}
```

- `-<rule number>`: Deactivate the rule SA<rule number>.
- `+<rule number>`: Activate the rule SA<rule number>.

Example

Rule 24 gets disabled for two lines (that means that it is not necessary here to write, for example, `nTest:=DINT#99`), and afterwards re-enabled:

```
{analysis -24}
nTest := 99;
iVar := INT#2;
{analysis +24}
```

Specifying multiple rules:

```
{analysis -10, -24, -18}
```

Attribute `{attribute 'analysis' := '...'}`

You can use the attribute `{attribute 'analysis' := '<sign><rule number>'}` in the declaration part for enabling or disabling particular rules for a complete programming object:

Syntax

```
{attribute 'analysis' := '<sign><rule number>|,<further rule numbers separated by commas>'}
```

Example

Rules SA0033 and SA0031 get disabled for the complete structure:

```
{attribute 'analysis' := '-33, -31'}
TYPE My_Structure :
STRUCT
    iLocal : INT;
    uiLocal : UINT;
    udiLocal : UDINT;
END_STRUCT
END_TYPE
```

Rule SA0100 is disabled for the array:

```
{attribute 'analysis' := '-100'}
big: ARRAY[1..10000] OF DWORD;
```

Creating an Archive in Your Controller on Download

Overview

Creating an archive in your controller allows you to have the complete project in your controller which is useful for maintenance tasks. The following procedure describes how to create a project archive automatically with the download procedure.

You can manually create a project archive be executing the **File > Source download...** command as described in the *Menu Commands online help* (see Menu Commands, Online Help).

Creating an Archive on Your Controller

To create an archive on your controller with the download procedure, perform the following three main tasks.

1.) Configure the sources to be downloaded.

To define the elements to add to the archive, proceed as follows:

Step	Action
1	Execute the command **Project > Project Settings**.
2	In the **Project Settings** dialog box, select **Source Download > Additional Files...** The following options can be selected: Option to connect with the uploaded project: • **Download information files** Options which are needed to connect with the same EcoStruxure Machine Expert version (for example, on another PC): • **Download information files** • **Referenced devices** • **Referenced libraries** • **Visualization Profile**

For further information, refer to the **Project Settings - Source Download** description (see Menu Commands, Online Help).

2.) Download the project (sources) to the controller.

To archive your project in your controller, proceed as follows:

Step	Action	Comment
1	Execute the command **Online > Login** to connect to the controller, page 237.	A message is displayed.
2	Select **Online > Source download to connected device**.	The project is archived and downloaded to the controller.

3.) Upload the project (sources) from the controller by executing the **Online > Source Upload...** command (see Menu Commands, Online Help).

Logic Editors

What’s in This Part

Common Features of Graphic Editors 266
FBD/LD/IL Editor 268
Ladder Editor 303
Continuous Function Chart (CFC) Editor 310
Sequential Function Chart (SFC) Editor 330
Structured Text (ST) Editor 356

Common Features of Graphic Editors

What's in This Chapter

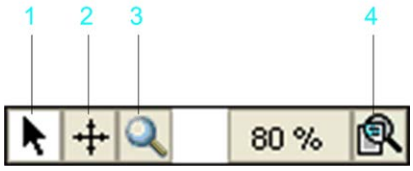
Common Features of Graphic Editors 266

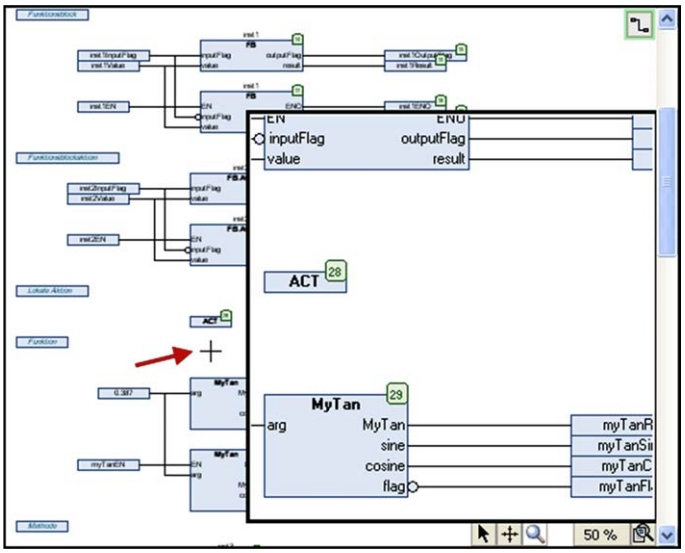
Common Features of Graphic Editors

Zooming, Panning, Magnifier Tools

The following zooming, panning, magnifier tools are available in the editors of all graphic programming languages.

The graphical editors for FBD, LD, CFC, and SFC provide a toolbar in the lower right corner of the editor window:



Item number	Name	Description
1	**Back to standard editing mode**	The pointer gets the usual shape of an arrow and you get back into the normal editing mode, in which you can select and edit elements in the editor window.
2	**Panning tool**	The pointer gets the shape of crossed arrows. You can click somewhere in the editor window and - while keeping the mouse-button pressed - shift the visible area of the chart within the window.
3	**Magnifier tool**	This function is useful when you have zoomed down the displayed chart to less than 100%. It opens a subwindow in the lower right corner of the editor window. As long as you move the pointer over your chart, this subwindow shows the respective part of the chart in 100% size. If you now click the window, the subwindow closes and that segment of the chart which was shown in the subwindow, is displayed in 100% size. So, if you want to keep the previously set zoom factor, use the **Back to standard editing mode** button to get back to the normal editing mode. 
4	**Zooming tool**	The zoom button opens a submenu where you can select one of the given zoom factors for the chart. To specify another value, click the **...** button to open an edit dialog box. The current zoom factor is displayed to the left of the button.

To zoom by mouse wheel, press the **Ctrl** key while scrolling the mouse wheel. The current zoom level is increased or decreased in steps of 10 percent.

You can drag function block declarations from the declaration part in the FBD and LD graphical editors to the editor view. To do this, select the full declaration (variable name and data type) and drag it to a suitable position in the editor view. In the ladder diagram, you can also drag Boolean declarations to the editor and insert them as contacts.

FBD/LD/IL Editor

What's in This Chapter

Information on the FBD/LD/IL Editor 268
FBD/LD/IL Elements 291
LD Elements 301

Information on the FBD/LD/IL Editor

FBD/LD/IL Editor

Overview

A combined editor is available for editing POUs in the languages FBD (function block diagram, page 268), LD (ladder diagram, page 269), and IL (instruction list, page 269).

Activate the IL implementation language in the **Tools > Options** dialog box, category **FBD, LD and IL editor**, tab (see Menu Commands, Online Help) **IL**.

Therefore, a common set of commands and elements is used and an automatic internal conversion between the three languages is done. In offline mode, the programmer can switch between editor views (**View**).

Keep in mind that there are some special elements which cannot be converted and thus will only be displayed in the appropriate language. Also, there are some constructs which cannot be converted unambiguously between IL and FBD and therefore will be normalized at a conversion back to FBD; specifically, negation of expressions and explicit/implicit output assignments.

You can define the behavior, look, and menus of the FBD/LD/IL editor in the **Customize** and **Options** dialog boxes. You also have options to define the display of comments and addresses.

The editor opens in a bipartite window. When you edit an object programmed in FBD/LD/IL, the upper part contains a declaration editor, page 373, the lower part contains a coding area.

The programming language for a new object is specified when you create the object.

For further information refer to:

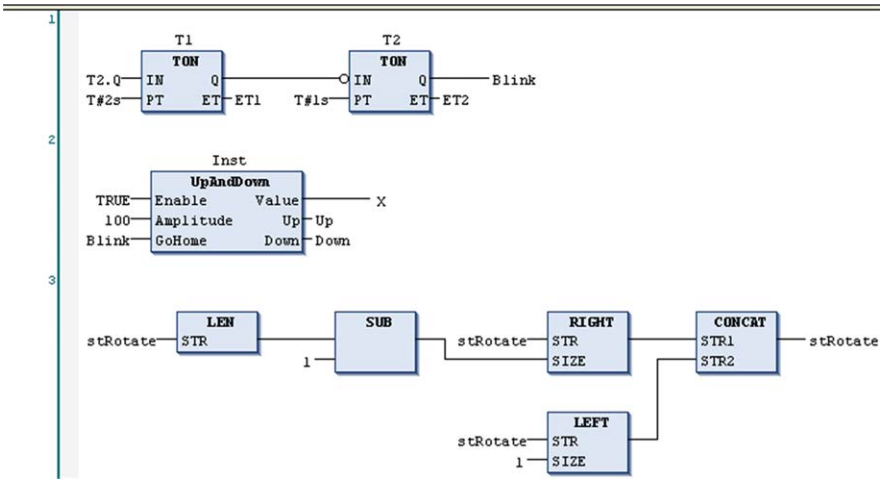
- *Working in the FBD and LD Editor View*, page 274
- *Working in the IL Editor View*, page 278

Function Block Diagram (FBD) Language

Overview

The Function Block Diagram is a graphically oriented programming language. It works with a list of networks. Each network contains a graphical structure of boxes and connection lines which represents either a logical or arithmetic expression, the call of a function block, a jump, or a return instruction.

FBD networks



Ladder Diagram (LD) Language

Overview

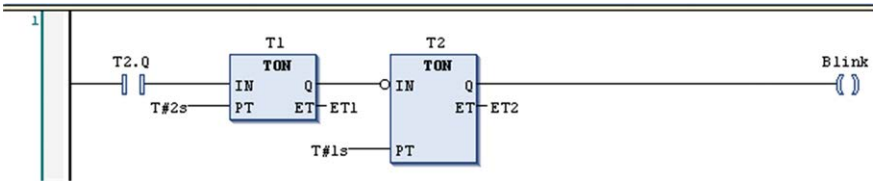
The Ladder Diagram is a graphics-oriented programming language which resembles the structure of an electric circuit.

On the one hand, the Ladder Diagram is suitable for constructing logical switches, on the other hand it also allows you to create networks as in FBD. Therefore, the LD is useful for controlling the call of other POUs.

The Ladder Diagram consists of a series of networks, each being limited by a vertical current line (power rail) on the left. A network contains a circuit diagram made up of contacts, coils, optionally additional POUs (boxes), and connecting lines.

On the left side, there is 1 or a series of contacts passing from left to right the condition ON or OFF which corresponds to the boolean values TRUE and FALSE. To each contact a boolean variable is assigned. If this variable is TRUE, the condition will be passed from left to right along the connecting line. Otherwise, OFF will be passed. Thus, the coil or coils, which is/are placed in the right part of the network, receive an ON or OFF coming from left. Correspondingly the value TRUE or FALSE will be written to an assigned boolean variable.

Ladder Diagram network.



Instruction List (IL) Language

Overview

The instruction list (IL)is an assembler-like IEC 61131-3 conformal programming language.

This language supports programming based on an accumulator. The IEC 61131-3 operators are supported as well as multiple inputs / multiple outputs, negations, comments, set / reset of outputs and unconditional / conditional jumps.

Each instruction is primarily based on the loading of values into the accumulator by using the LD operator. After that the operation is executed with the first parameter taken out of the accumulator. The result of the operation is available in the accumulator, from where you should store it with the ST instruction.

In order to program conditional executions or loops, IL supports both comparing operators such as EQ, GT, LT, GE, LE, NE and jumps. The latter can be unconditional (JMP) or conditional (JMPC / JMPCN). For conditional jumps, the value of the accumulator is referenced for TRUE or FALSE.

Syntax

An instruction list (IL) consists of a series of instructions. Each instruction begins in a new line and contains an operator and, depending on the type of operation, 1 or more operands separated by commas. You can extend the operator by a modifier.

In a line before an instruction, there can be an identification mark (label) followed by a colon (:) (`m1:` in the example shown below). A label can be the target of a jump instruction (`JMPC m1` in the example shown below).

Place a comment as last element of a line.

You can insert empty lines between instructions.

Example

```
LD            BVar1
ST            tonInst1.IN
CAL           tonInst1(
          PT:=t1,
          ET=>tOut2)
LD            toninst1.Q
JMPC          mark1
ST            tonInst2.IN

_____________________________________________________________________

mark1:
LD            iVar2
ADD           230
```

For further information, refer to:

- *Working in the IL Editor View*, page 278
- *Modifiers and operators in IL*, page 270

Modifiers and Operators in IL

Modifiers

You can use the following modifiers in Instruction List, page 269.

C	with JMP, CAL, RET:	The instruction will only be executed if the result of the preceding expression is TRUE.
N	with JMPC, CALC, RETC:	The instruction will only be executed if the result of the preceding expression is FALSE.

N	with operators according to the *Operators* table below (*N* in *Modifiers* column)	Negation of the operand (not of the accumulator).
(	with operators according to the *Operators* table below (*(* (left parenthesis) in *Modifiers* column)	To be used for complex operands. For details, refer to the list of use cases for complex operands, page 272.

NOTE: Generally, it is not good practice to use the statement CALC (/RETC/ JMPC) directly after an STN, S or R operator, as those instructions arbitrarily modify the value of the accumulator and thus could lead to difficult-to-find programming errors.

Operators

The table shows which operators can be used in combination with the specified modifiers.

The accumulator stores the current value, resulting from the preceding operation.

Operator	Modifiers	Meaning	Example
LD	N	Loads the (negated) value of the operand into the accumulator.	`LD iVar`
ST	N	Stores the (negated) content of the accumulator into the operand variable.	`ST iErg`
S	–	Sets the operand (type BOOL) to TRUE when the content of the accumulator is TRUE.	`S bVar1`
R	–	Sets the operand (type BOOL) to FALSE when the content of the accumulator is TRUE.	`R bVar1`
AND	N,(	Bitwise AND of the accumulator and the (negated) operand.	`AND bVar2`
OR	N,(	Bitwise OR of the accumulator and the (negated) operand.	`OR xVar`
XOR	N,(	Bitwise exclusive OR of the accumulator and the (negated) operand.	`XOR N, (bVar1, bVar2)`
NOT	–	Bitwise negation of the content of the accumulator.	–
ADD	(	Addition of accumulator and operand, result is copied to the accumulator.	`ADD iVar1`
SUB	(	Subtraction of accumulator and operand, result is copied to the accumulator.	`SUB iVar2`
MUL	(	Multiplication of accumulator and operand, result is copied to the accumulator.	`MUL iVar2`
DIV	(	Division of accumulator and operand, result is copied to the accumulator.	`DIV 44`
GT	(	Verifies if accumulator is greater than the operand, result (BOOL) is copied into the accumulator; >	`GT 23`
GE	(	Verifies if accumulator is greater than or equal to the operand, result (BOOL) is copied into the accumulator; >=	`GE iVar2`
EQ	(	Verifies if accumulator is equal to the operand, result (BOOL) is copied into the accumulator; =	`EQ iVar2`
NE	(	Verifies if accumulator is not equal to the operand, result (BOOL) is copied into the accumulator; <>	`NE iVar1`
LE	(	Verifies if accumulator is less than or equal to the operand, result (BOOL) is copied into the accumulator; <=	`LE 5`

Operator	Modifiers	Meaning	Example
LT	(	Verifies if accumulator is less than operand, result (BOOL) is copied into the accumulator; <	`LT cVar1`
JMP	C, CN	Unconditional (conditional) jump to the label • JMP = unconditional jump • JMPC = conditional jump if accumulator is TRUE • JMPCN = conditional jump if accumulator is FALSE	`JMP next`
CAL	C, CN	Unconditional (conditional) call of a PROGRAM or FUNCTION_BLOCK • CAL = unconditional call • CALC = conditional call if accumulator is TRUE • CALCN = conditional call if accumulator is FALSE	`CAL prog1`
RET	C, CN	Unconditional (conditional) return of the POU and jump back to the calling POU • RET = unconditional return • RETC = conditional return if accumulator is TRUE • RETCN = conditional return if accumulator is FALSE	`RET`
)	–	Evaluate deferred operation	–

See also IEC operators, page 678 and Work in IL editor, page 278 for how to use and handle multiple operands, complex operands, function / method / function block / program / action calls and jumps.

Example

Example IL program using some modifiers:

```
LD      TRUE        load TRUE to accumulator
ANDN    bVar1       execute AND with negative value of bVar1
JMPC    m1          if accum. is TRUE, jump to label "m1"
LDN     bVar2       store negated value of bVar2...
ST      bRes        ... in bRes
```

```
m1:
LD      bVar2       store value of bVar2...
ST      bRes        ... in bRes
```

Using Operands

The table lists use cases for operands:

Use case	Description	Examples

Several operands for one operator	Options
	• Enter the operands into consecutive rows, separated by commas in the 2nd column.
	• Repeat the operator in consecutive rows.
		• Option 1:
		`LD 2`
		`ADD 3,`
		`4,`
		`6`
		`ST iVAR`
		• Option 2:
		`LD 2`
		`ADD 3`
		`ADD 4`
		`ADD 6`
		`ST iVAR`

Complex operands	For a complex operand, enter the left parenthesis (in the first column.	A string is rotated by one character each cycle:
		`LD stRotate`
		`RIGHT( stRotate`

Use case	Description	Examples

	Enter the right parenthesis in the first column in a separate row followed by the operand entries in the following rows.	`LEN`
		`SUB 1`
		`)`
		`CONCAT( stRotate,`
		`LEFT 1`
		`)`
		`ST stRotate`

Function block call, program call	Row 1:
	• Column 1:
	Operator `CAL` or `CALC`
	• Column 2:
	Name of the function block instance or the program and left parenthesis `(`. If no parameters follow, the right parenthesis `)` is entered here.
	Following rows:
	• Column 1:
	Parameter name followed by:
	◦ `:=` for input parameters
	◦ `=>` for output parameters
	• Column 2:
	Parameter value followed by a comma `,` if followed by further parameters.
	Enter the right parenthesis `)` after the last parameter.
	According to the IEC standard complex expressions are not allowed here. Assign such constructs to the function block or the program before it is called.
		`CAL POUToCall(`
		`iCounter:=1,`
		`iDecrement:=1000,`
		`wError:=wResult)`
		
		`LD POUToCall.bError,`
		`ST bErr`

Function call	Row 1:
	• Column 1:
	`LD`
	• Column 2:
	Input variable
	Row 2:
	• Column 1:
	Function name
	• Column 2:
	Further input parameters, separated by commas
	The return value is written to the accumulator.
	Row 3:
	• Column 1:
	`ST`
	• Column 2:
	Variable to which the return value is written
		`LD X7`
		`GeomAverage 25`
		`ST Ave`

Use case	Description	Examples
Action call	Similar to function block call or program call. The name of the action is appended to the name of the function block instance or the program.	`CAL POU1.ResetAction`
Jump	• Column 1: Operator `JMP` or `JMPC` • Column 2: Name of the jump label of the destination network. • Unconditional jump: The preceding instruction sequence must end with one of the following commands: ◦ `ST` ◦ `STN` ◦ `S` ◦ `R` ◦ `CAL` ◦ `RET` ◦ `JMP` • Conditional jump: The execution of the jump depends on the loaded value.	`LD BVar1` `JMPC Label1`

Working in the FBD and LD Editor

Overview

Networks are the basic entities in FBD and LD programming. Each network contains a structure that displays a logical or an arithmetical expression, a POU (function, program, function block call, and so on), a jump, a return instruction.

When you create a new object, the editor window automatically contains 1 empty network.

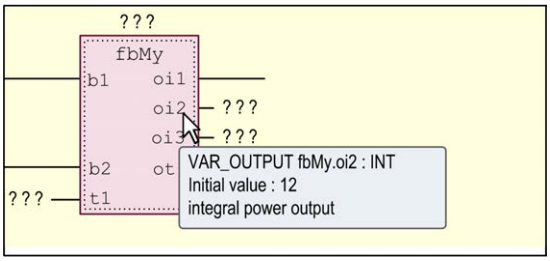
Refer to the general editor settings in the **Options** dialog box, category **FBD/LD/IL** for potential editor display options.

Tooltip

Tooltips contain information on variables or box parameters.

The cursor being placed on the name of a variable or box parameter will prompt a tooltip. It shows the respective type. In case of function block instances the scope, name, datatype, initial value, and comment will be displayed. For IEC operators `SEL`, `LIMIT`, and `MUX` a short description on the inputs will display. If defined, the address and the symbol comment will be shown as well as the operand comment (in quotation marks in a second line).

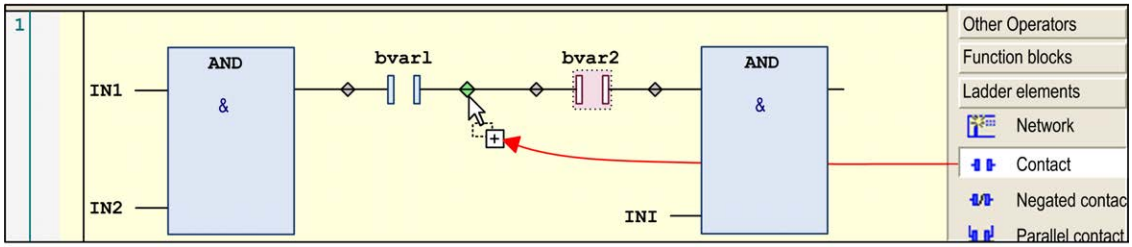
Example: Tooltip on a POU output



Inserting, Arranging, and Replacing Elements

- The commands for working in the editor are by default available in the FBD/LD/IL menu, page 286. Frequently used commands are also available in the contextual menu. It depends on the current cursor position or the current selection (multiselection possible, see below Selecting, page 276) which elements can be inserted via menu command.
- You can also drag elements with the mouse from the **ToolBox**, page 291 to the editor window or from one position within the editor to another (drag and drop). For this purpose select the element to be dragged by a mouse-click, keep the mouse-button pressed and drag the element into the respective network in the editor view. As soon as you have reached the network, all possible insert positions for the respective type of element will be indicated by gray position markers. When you move the mouse-cursor on one of these positions, the marker will change to green and you can release the mouse-button in order to insert the element at that position.
- If you drag a function block, an operator, or a network from the **ToolBox**, page 291 onto the up arrow or down arrow on the left-hand side of the editor, a new network is automatically created above or below the existing element.
- To replace an element by another one, draw another element on the position.
- To reposition an input connection or an output connection of a box, select the connection pin directly at the box and put it to the desired other position at the box via drag and drop.

Insert positions in LD editor



- You can use the cut, copy, paste, and delete commands, available in the **Edit** menu, to arrange elements. You can also copy an element by drag and drop: select the element within a network by a mouse-click, press the CTRL key and while keeping the mouse button and the key pressed, drag the element to the target position. As soon as that is reached (green position marker), a plus-symbol will be added to the cursor symbol. Then, release the mouse-button to insert the element.
- For possible cursor positions, refer to *Cursor Positions in FBD, LD, and IL*, page 283.
- Inserting of EN/ENO boxes is handled diversely in the FBD and LD editor.

  Refer to the description of the **Insert Box** command for further information (Inserting of EN/ENO boxes is not supported in the IL editor).

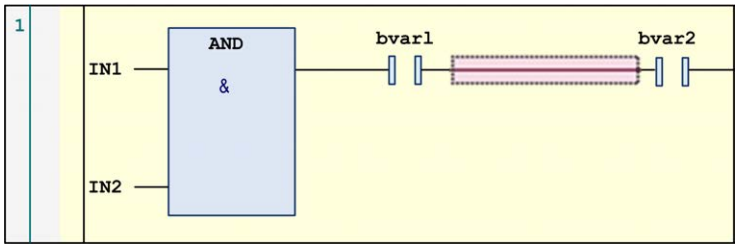
Navigating in the Editor

- You can use the ARROW keys to jump to the next or previous cursor position, page 283. This is also possible between networks. The navigation with the ← and → key follows the signal flow which is normally from left to right and vice versa. In case of line breaks, the following cursor position can also be left under the currently marked position. If you press the ↑ or ↓ key the selection jumps to the next neighbor above or below the current position if this neighbor is in the same logical group (for example, a pin of a box). If no such group exists, it jumps to the nearest neighbor element above or below. Navigation through parallel connected elements is performed along the first branch.
- Press the HOME key to jump to the first element. Press the END key to jump to the last element of the network.
- Use the TAB key to jump to the next or previous cursor position, page 283 within a network.
- Press CTRL + HOME to scroll to the begin of the document and to mark the first network.
- Press CTRL + END to scroll to the end of the document and to mark the last network.
- Press PAGE UP to scroll up 1 screen and to mark the topmost rectangle.
- Press PAGE DOWN to scroll down 1 screen and to mark the topmost rectangle.

Selecting

- You can select an element, also a network, via taking the respective cursor position by a mouse-click or using the arrow or tab keys. Selected elements are indicated as red-shaded. Also refer to *Cursor Positions in FBD, LD, and IL*, page 283.
- In the LD editor, you can also select the lines between elements in order to execute commands, for example, for inserting a further element at that position.

Selected line in LD editor

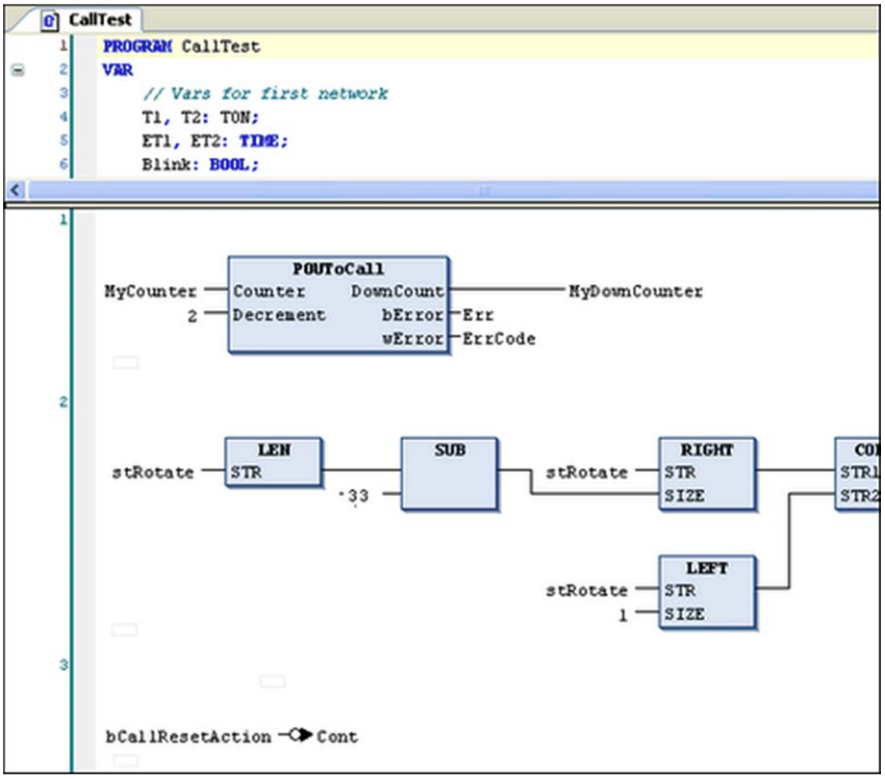


- Multi-selection of elements or networks is possible by keeping pressed the CTRL key while selecting the desired elements one after the other.

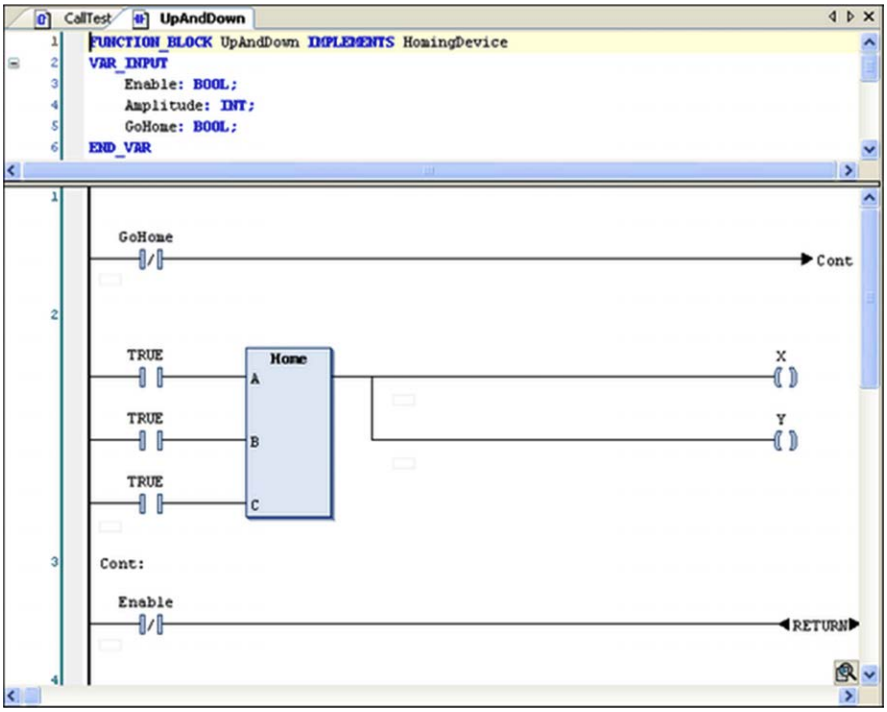
Open a Function Block

If a function block is added to the editor, you can open this block with a double-click. Alternatively, you can use the command **Browse - Go To Definition** from the contextual menu.

FBD editor



LD editor



For information on the languages, refer to:

- *Function Block Diagram - FBD*, page 268
- *Ladder Diagram - LD*, page 269

Working in the IL Editor

Overview

The IL Instruction List, page 269 editor is a table editor. The network structure of FBD or LD programs is also represented in an IL program. Basically, one network, page 292 is sufficient in an IL program, but considering switching between FBD, LD and IL you also can use networks for structuring an IL program.

Activate the IL implementation language in the **Tools > Options** dialog box, category **FBD, LD and IL editor**, tab (see Menu Commands, Online Help) **IL**.

Tooltip

Tooltips contain information on variables or box parameters.

Refer to *Working in the FBD and LD Editor View*, page 274.

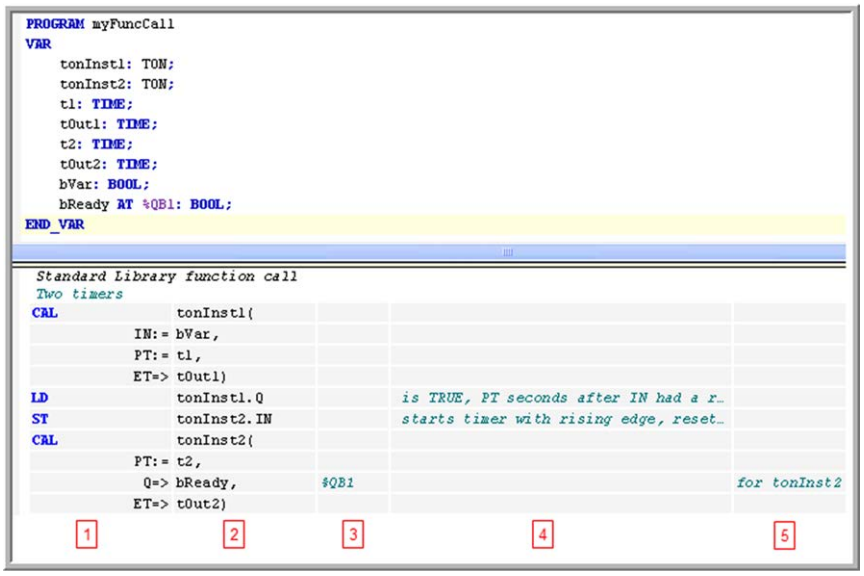
Inserting and Arranging Elements

- The commands for working in the editor are available in the **FBD/LD/IL** menu. Frequently used commands are also available in the contextual menu.
- Programming units that are elements are inserted each at the current cursor position via the **Insert** commands, available in the **FBD/LD/IL** menu.
- You can drag a **Network** element from the **ToolBox**, page 291 onto the up arrow or down arrow on the left-hand side of the editor. A new network is then created above or below the existing element.
- You can use the cut, copy, paste, and delete commands, available in the **Edit** menu, to arrange elements.
- See also some information on the programming language Instruction List - IL, page 269.
- Operators with EN/ENO functionality can be inserted only within the FBD and LD editor.

This chapter explains how the table editor is structured, how you can navigate in the editor and how to use complex operands, calls and jumps.

Structure of the IL Table Editor

IL table editor



Each program line is written in a table row, structured in fields by the table columns:

Col-umn	Contains...	Description
1	operator	This field contains the IL operator (LD, ST, CAL, AND, OR, and so on) or a function name. In case of a function block call, the respective parameters are also specified here. Enter the prefix field `:=` or `=>`. For further information, refer to *Modifiers and Operators in IL*, page 270.
2	operand	This field contains exactly one operand or a jump label. If more than one operand is needed (multiple/extensible operators AND A, B, C, or function calls with several parameters), write them in the following lines and leave the operator field empty. In this case, add a parameter-separating comma. In case of a function block, program or action call, add the opening and/or closing brackets.
3	address	This field contains the address of the operand as defined in the declaration part. You cannot edit this field. Use the option **Show symbol address** to switch it on or off.
4	symbol comment	This field contains the comment as defined for the operand in the declaration part. You cannot edit this field. Use the option **Show symbol address** to switch it on or off.
5	operand comment	This field contains the comment for the current line. It is editable and can be switched on or off via option **Show operand comment**.

Navigating in the Table

- UP and DOWN arrow keys: Moving to the field above or below.
- TAB: Moving within a line to the field to the right.
- SHIFT + TAB: Moving within in a line to the field to the left.
- SPACE: Opens the currently selected field for editing. Alternatively, performs a further mouse-click on the field. If applicable, the input assistant will be available via the **...** button. You can close a currently open edit field by pressing ENTER, confirming the current entries, or by pressing ESC canceling the made entries.
- CTRL + ENTER: Enters a new line below the current one.
- DEL: Removes the current line that is where you have currently selected any field.
- **Cut**, **Copy**, **Paste**: To copy 1 or several lines, select at least 1 field of the line or lines and execute the **Copy** command. To cut a line, use the **Cut** command. **Paste** will insert the previously copied/cut lines before the line where currently a field is selected. If no field is selected, they will be inserted at the end of the network.
- CTRL + HOME scrolls to the begin of the document and marks the first network.
- CTRL + END scrolls to the end of the document and marks the last network.
- PAGE UP scrolls up 1 screen and marks the topmost rectangle.
- PAGE DOWN scrolls down 1 screen and marks the topmost rectangle.

Multiple Operands (Extensible Operators)

If the same operator, page 270 is used with multiple operands, 2 ways of programming are possible:

- The operands are entered in subsequent lines, separated by commas, for example:

```
LD      7
ADD     2,
        4,
        7
ST      iVar
```

- The instruction is repeated in subsequent lines, for example:

```
LD      7
ADD     2
ADD     4
ADD     7
ST      iVar
```

Complex Operands

If a complex operand is to be used, enter an opening bracket before, then use the following lines for the particular operand components. Below them, in a separate line, enter the closing bracket.

Example: Rotating a string by 1 character at each cycle.

Corresponding ST code:

```
stRotate := CONCAT(RIGHT(stRotate, (LEN(stRotate) - 1)),
(LEFT(stRotate, 1)));
LD          stRotate
RIGHT(      stRotate
LEN
SUB         1
)
CONCAT(     stRotate
LEFT        1
)
ST          stRotate
```

Function Calls

Enter the function name in the operator field. Give the (first) input parameter as an operand in a preceding LD operation. If there are further parameters, give the next one in the same line as the function name. You can add further parameters in this line, separated by commas, or in subsequent lines.

The function return value will be stored in the accumulator. The following restriction concerning the IEC standard applies.

> **NOTE:** A function call with multiple return values is not possible. Only 1 return value can be used for a succeeding operation.

Example: Function *GeomAverage*, which has 3 input parameters, is called. The first parameter is given by *X7* in a preceding operation. The second one, *25*, is given with the function name. The third one is given by variable *tvar*, either in the same line or in the subsequent one. The return value is assigned to variable *Ave*.

Corresponding ST code:

```
Ave := GeomAverage(X7, 25, tvar);
```

Function call in IL:

```
LD              X7
GeomAverage     25
                tvar
ST              Ave
```

Function Block Calls and Program Calls

Use the CAL- or CALC operator, page 270. Enter the function block instance name or the program name in the operand field (second column) followed by the opening bracket. Enter the input parameters each in one of the following lines:

Operator field: Parameter name

Prefix field:

- `:=` for input parameters
- `=>` for output parameter

Operand field: Current parameter

Postfix field:

- `,` if further parameters follow
  `)` after the last parameter
- `()` in case of parameter-less calls

Example: Call of *POUToCAll* with 2 input and 2 output parameters.

Corresponding ST code:

```
POUToCall(Counter := iCounter, iDecrement:=2, bError=>bErr,
wError=>wResult);
```

Program call in IL with input and output parameters:

```
1  PROGRAM IL_EXAMPLE
2  VAR
3      bErr: BOOL;
4      wResult: WORD;
5  END_VAR
6
```

```
1
   CAL          POUToCall(
          Counter := iCounter
       iDecrement := 2
           wError=> wResult)
   LD           POUToCall.bError
   ST           bErr
```

It is not necessary to use all parameters of a function block or program.

NOTE: Complex expressions cannot be used. These must be assigned to the input of the function block or program before the call instruction.

Action Call

To be performed like a function block or program call, the action name is to be appended to the instance name or program name.

Example: Call of action *ResetAction*.

Corresponding ST code:

```
Inst.ResetAction();
```

Action call in IL:

```
CAL      Inst.ResetAction()
```

Method Call

To be performed like a function call, the instance name with appended method name is to be entered in the first column (operator).

Example: Call of method `Home`.

Corresponding ST code:

```
Z := IHome.Home(TRUE, TRUE, TRUE);
```

Method call in IL:

```
LD              TRUE
IHome.Home      TRUE
                TRUE
ST              Z
```

Jump

A jump, page 294 is programmed by `JMP` in the first column (operator) and a label name in the second column (operand). The label is to be defined in the target network in the label, page 294 field.

The statement list preceding the unconditional jump has to end with one of the following commands: ST, STN, S, R, CAL, RET, or another JMP.

This is not the case for a conditional jump, page 294. The execution of the jump depends on the value loaded.

Example: Conditional jump instruction; in case `bCallRestAction` is TRUE, the program should jump to the network labeled with `Cont`.

Conditional jump instruction in IL:

```
LDN         bCallResetAction
JMPC        Cont
```

Cursor Positions in FBD, LD, and IL

IL Editor

This is a text editor, structured in form of a table. Each table cell is a possible cursor position. Also refer to *Working in the IL Editor View*, page 278.

FBD and LD Editor

These are graphic editors, see below examples (1) to (15) showing the possible cursor positions: text, input, output, box, contact, coil, return, jump, line between elements and network.

Actions such as cut, copy, paste, delete, and other editor-specific commands refer to the current cursor position or selected element. See *Working in the FBD and LD Editor*, page 274.

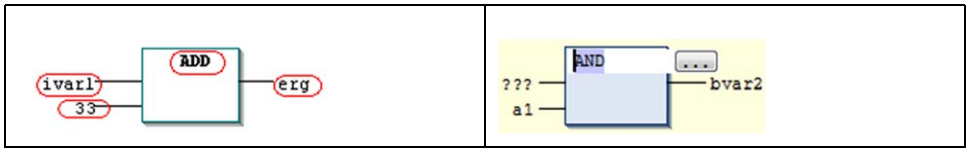
Basically, in FBD a dotted rectangle around the respective element indicates the current position of the cursor. Additionally, texts and boxes become blue- or red-shadowed.

In LD, coils and contacts become red-colored as soon as the cursor is positioned on.

The cursor position determines which elements are available in the contextual menu for getting inserted, page 275.

Possible Cursor Positions

(1) Every text field

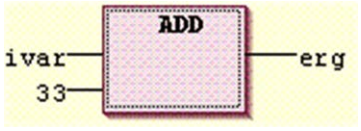


In the left image, the possible cursor positions are marked by a red-frame. The right image shows a box with the cursor being placed in the *AND* field. Keep in mind the possibility to enter addresses instead of variable names if configured appropriately in the **FBD, LD and IL editor Options** dialog box.

(2) Every input:



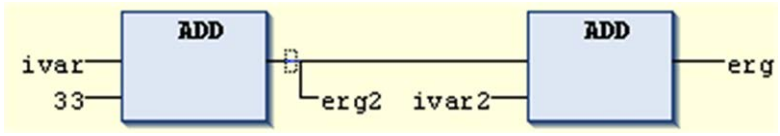
(3) Every operator, function, or function block:



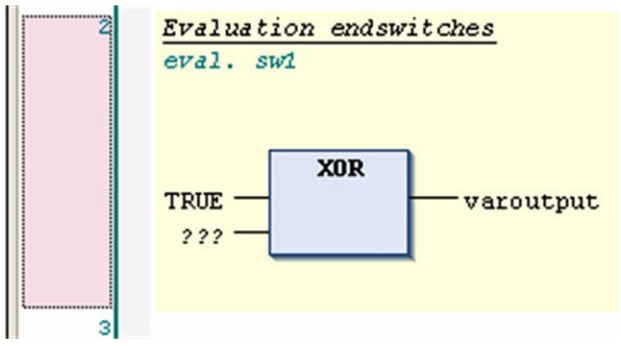
(4) Outputs if an assignment or a jump comes afterward:



(5) Just before the lined cross above an assignment before a jump or a return instruction:



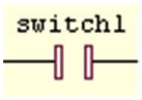
(6) The right-most cursor position in the network or anywhere else in the network besides the other cursor positions. This will select the whole network:



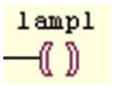
(7) The lined cross directly in front of an assignment:



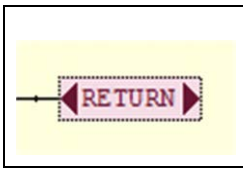
(8) Every contact:



(9) Every coil:



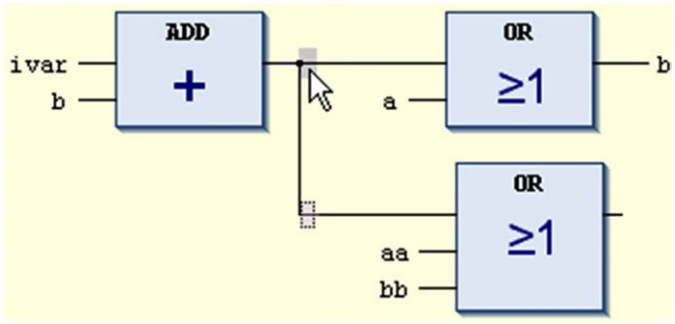
(10) Every return and jump:



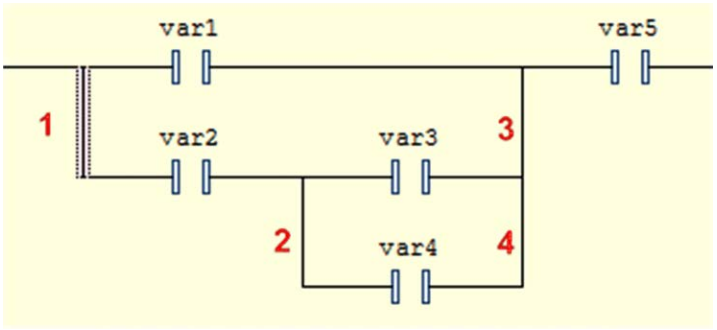
(11) The connecting line between the contacts and the coils:



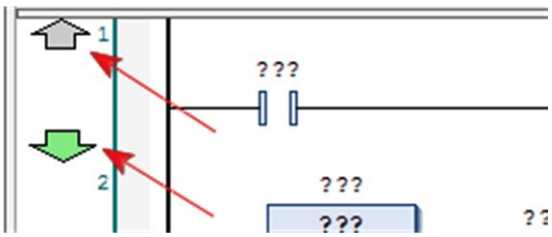
(12) Branch or subnetwork within a network:



(13) The connection line between parallel contacts (Pos. 1...4):

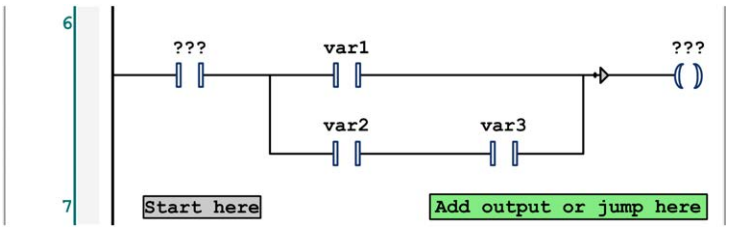


(14) Before or after a network:



You can add new networks on the left-most side of the editor. The insertion of a new network before an existing network is only possible before network 1.

(15) Begin or end of a network:



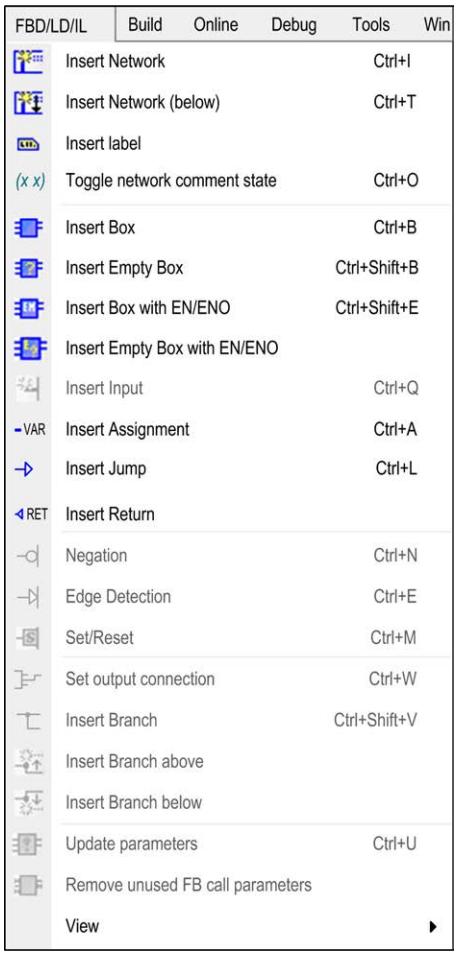
You can add contacts and function blocks at the begin of a network on the field **Start here**. You can add the elements return, jump, and coil at the end of a network on the field **Add output or jump here**.

FBD/LD/IL Menu

Overview

When the cursor is placed in the FBD/LD/IL Editor, page 268 window, the FBD/LD/IL menu is available in the menu bar, providing the commands for programming in the currently set editor view.

FBD/LD/IL menu in FBD editor view:



- For a description of the commands, refer to the chapter *FBD/LD/IL Editor Commands*.
- For configuration of the menu, refer to the description of the **Customize Menu**.

FBD/LD/IL Editor in Online Mode

Overview

In online mode, the FBD/LD/IL editor provides views for Monitoring, page 287 and for writing and forcing the values and expressions on the controller.

Debugging functionality (breakpoints, stepping, and so on) is available, see *Breakpoint or Halt Positions*, page 290.

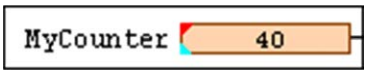
- For information on how to open objects in online mode, refer to the chapter *User Interface in Online Mode*, page 50.
- Keep in mind that the editor window of an FBD, LD, or IL object also includes the **Declaration Editor** in the upper part. Also refer to the chapter *Declaration Editor in Online Mode*, page 378.

Monitoring

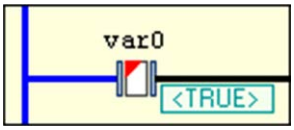
If the inline monitoring is not explicitly deactivated in the **Options** dialog box, it will be supplemented in FBD or LD editor by small monitoring windows behind each variable or by an additional monitoring column showing the actual values (inline monitoring). This is even the case for unassigned function block inputs and outputs.

The inline monitoring window of a variable shows a little red triangle in the upper left corner if the variable is currently forced, page 289, a blue triangle in the lower left corner if the variable is currently prepared for writing or forcing. In LD, for contacts and coils the currently prepared value (TRUE or FALSE) will be displayed down right below the element.

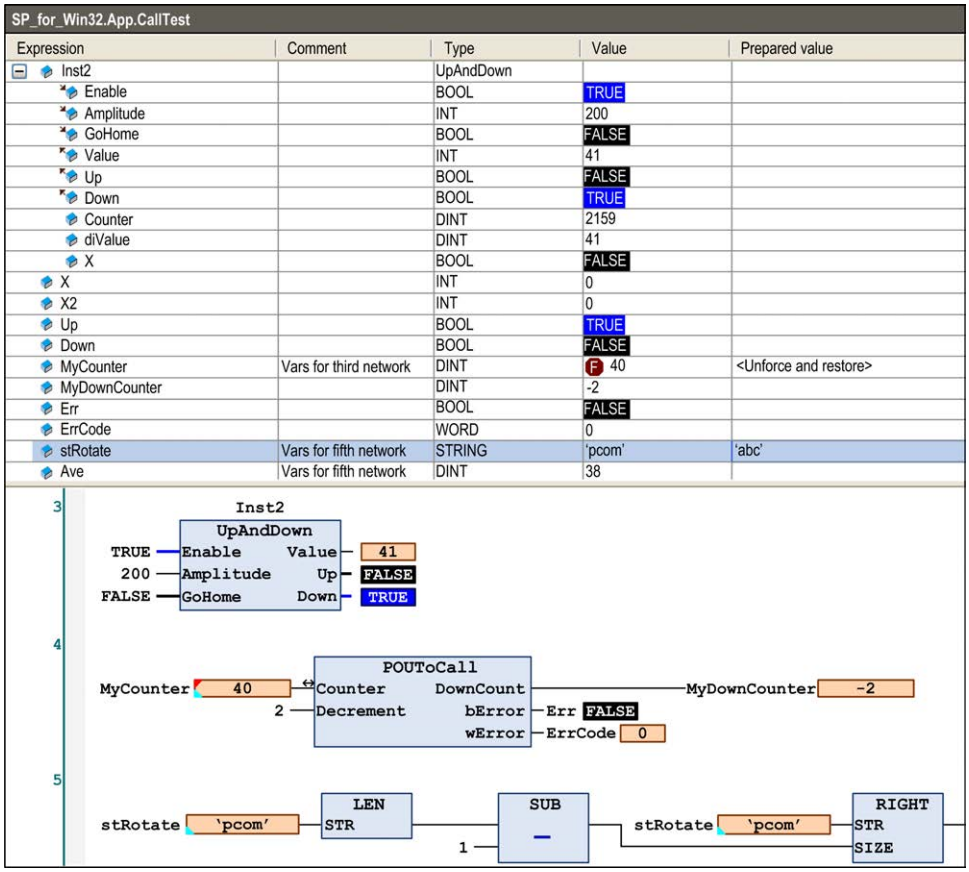
Example for a variable which is currently forced and prepared for releasing the force



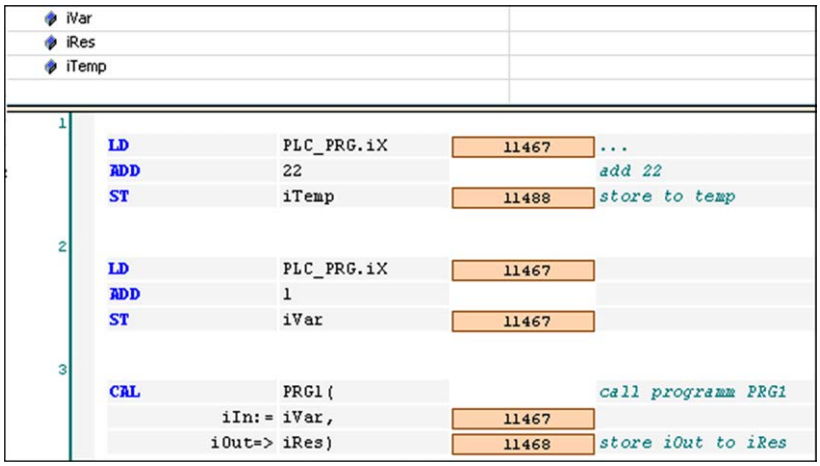
Example for a contact variable which is currently prepared to get written or forced with value TRUE



Online view of an FBD program



Online view of an IL program

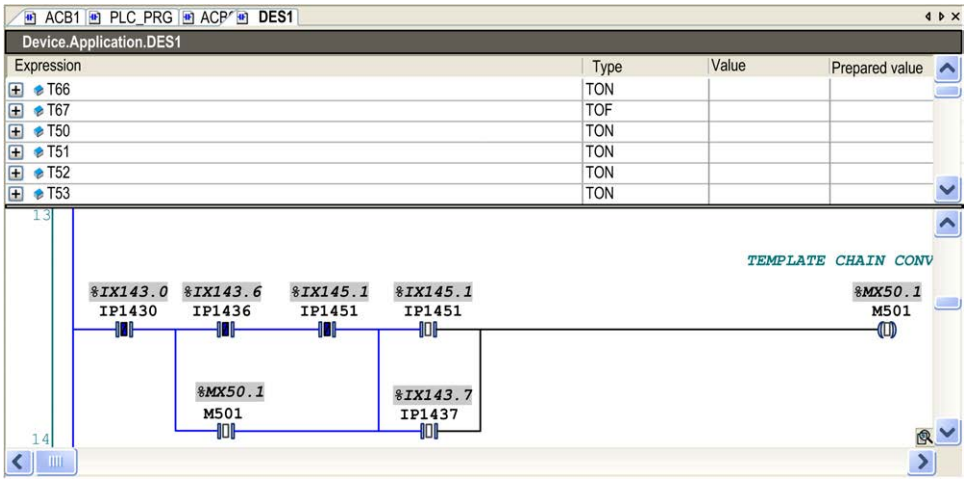


In online view, ladder networks have animated connections:

- Connections with value TRUE are displayed in bold blue.
- Connections with value FALSE are displayed in bold black.
- Connections with no known value or with an analog value are displayed in standard outline (black and not bold).

The values of the connections are calculated from the monitoring values.

Online view of an LD program

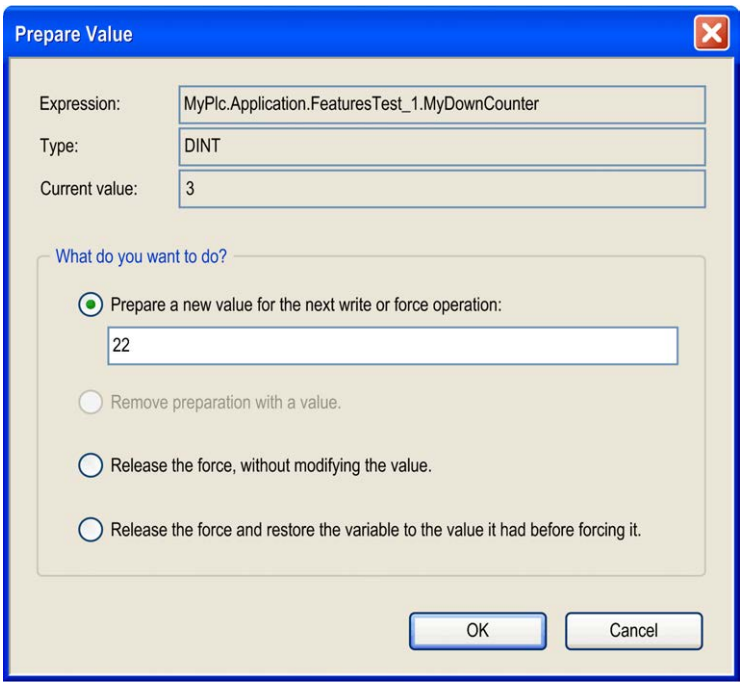


Open a function by double-click or execute the command **Browse - Go To Definition** from the contextual menu. Refer to the description of the *User Interface in Online Mode*, page 50 for further information.

Forcing/Writing of Variables

In online mode, you can prepare a value for forcing or writing a variable either in the declaration editor, page 378 or within the editor. Double-click a variable in the editor to open the following dialog box:

Dialog box **Prepare Value**



You find the name of the variable completed by its path within the device tree (**Expression**), its type, and current value. By activating the corresponding item, you can do the following:

- Preparing a new value which has to be entered in the edit field.
- Removing a prepared value.
- Releasing the forced variable.
- Releasing the forced variable and resetting it to the value it was assigned to just before forcing.

The selected action will be carried out on executing the menu command **Force values** (in the **Online** menu) or by pressing F7.

For information on how the current state of a variable (forced, prepared value) is indicated at the respective element in the network, refer to the section *Monitoring*, page 287.

Breakpoint or Halt Positions

Possible positions you can define for a breakpoint (halt position) for debugging purposes are those positions at which values of variables can change (statements), at which the program flow branches out, or at which another POU is called.

These are the following positions:

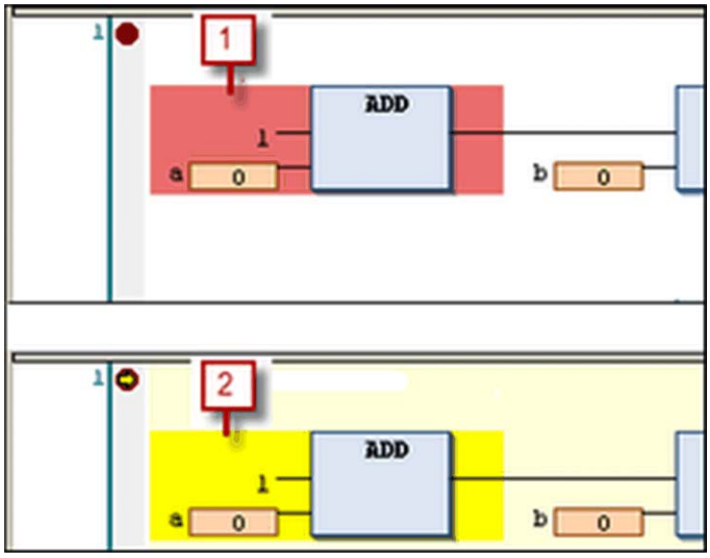
- On the network as a whole such that the breakpoint will be applied to the first possible position within the network.
- On a box, page 294, if this contains a statement. Therefore it is not possible on operator boxes like for example `ADD, DIV`. See the Note below.
- On an assignment.
- At the end of a POU at the point of return to the caller; in online mode, automatically an empty network will be displayed for this purpose. Instead of a network number, it is identified by `RET`.

  NOTE: You cannot set a breakpoint directly on the first box of a network. If, however, a breakpoint is set on the complete network, the halt position will automatically be applied to the first box.

For the currently possible positions, refer to the selection list within the **View > Breakpoints** dialog box.

A network containing any active breakpoint position is marked with the breakpoint symbol (red filled circle) right to the network number and a red-shaded rectangle background for the first possible breakpoint position within the network. Deactivated breakpoint positions are indicated by a non-filled red circle or a surrounding non-filled red rectangle.

Breakpoint set and breakpoint reached

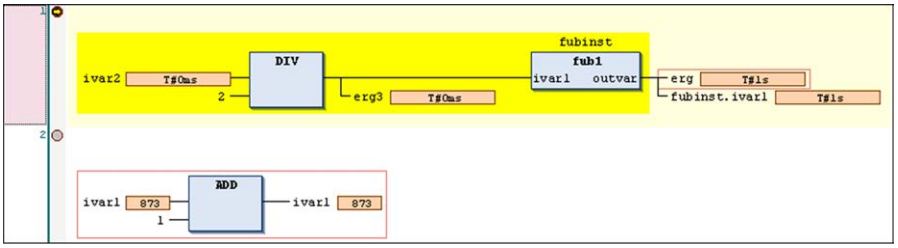


1 breakpoint set

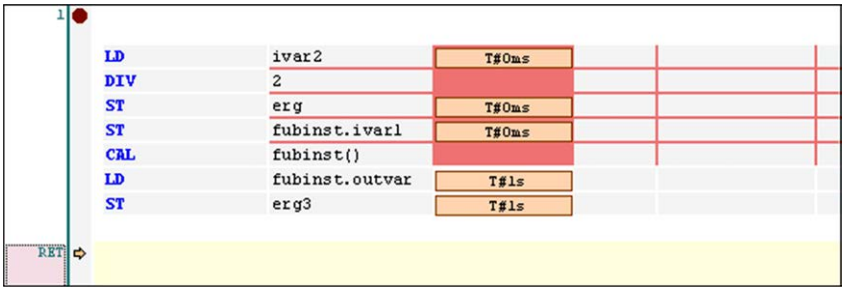
2 breakpoint reached

As soon as a breakpoint position is reached during stepping or program processing, a yellow arrow will be displayed in the breakpoint symbol and the red shaded area will change to yellow.

Halt positions shown in FBD



Halt position shown in IL



NOTE: A breakpoint will be set automatically in all methods which may be called. If an interface-managed method is called, breakpoints will be set in all methods of function blocks implementing that interface and in all derivative function blocks subscribing the method. If a method is called via a pointer on a function block, breakpoints will be set in the method of the function block and in all derivative function blocks which are subscribing to the method.

FBD/LD/IL Elements

FBD/LD/IL Toolbox

Overview

The FBD/LD/IL Editor, page 268 provides a toolbox which offers the programming elements for being inserted in the editor window by drag and drop. Open the toolbox by executing the command **ToolBox** which is in the **View** menu.

It depends on the currently active editor view which elements are available for inserting (see the respective description of the insert commands).

The elements are sorted in categories: **General** (general elements such as **Network, Assignment** and so on), **Boolean operators**, **Math operators**, **Other operators** (for example, `SEL, MUX, LIMIT`, and `MOVE`), **Function blocks** (for example, `R_TRIG, F_TRIG, RS, SR, TON, TOF, CTD, CTU`), **Ladder elements**, and **POUs** (user-defined).

The **POUs** category lists all POUs which have been defined below the same application as the FBD/LD/IL object which is open in the editor. If a POU has been assigned a bitmap in its properties, then this will be displayed before the POU name. Otherwise, the standard icon for indicating the POU type will be used. The list will be updated automatically when POUs are added or removed from the application.

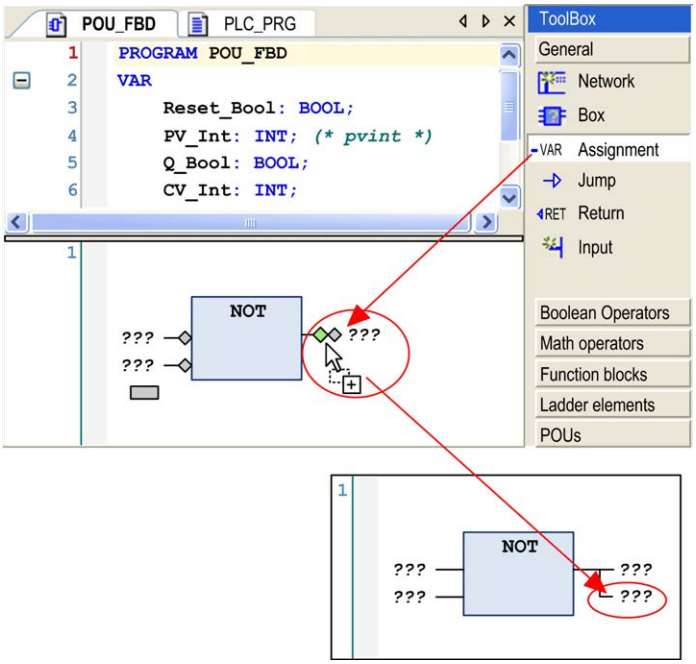
The category **Other operators** contains among `SEL, MUX, LIMIT`, and `MOVE` operators a conversion placeholder element. You can drag and drop this element to the desired position of the network. The conversion type is set automatically, dependent on the required type of the insert position. In some situations however the required conversion type cannot be determined automatically. Change the element manually in this case.

To unfold the category folders, click the button showing the respective category name. See in the following image: The category **General** is unfolded, the others

are folded. The image shows an example for inserting an **Assignment** element by drag and drop from the toolbox.

Only the section **General** in the toolbox is unfolded:

Insert from toolbox



To insert an element in the editor, select it in the toolbox by a mouse-click and bring it to the editor window by drag and drop. The possible insert positions will be indicated by position markers, which appear as long as the element is drawn - keeping the mouse button pressed - across the editor window. The nearest possible position will light up green. When you release the mouse button, the element will be inserted at the green position.

If you drag a box element on an existing box element, the new one replaces the old one. If inputs and outputs already have been assigned, those will remain as defined, but they will not be connected to the new element box.

Network in FBD/LD/IL

Overview

A network is the basic entity of an FBD, page 268 or LD, page 269 program. In the FBD/LD editor, the networks are arranged in a vertical list. Each network is designated on the left side by a serial network number and has a structure consisting of either a logical or an arithmetic expression, a program, function or function block call, and possibly jump or return instructions.

The IL Editor, page 269, due to the common editor base with the FBD and LD editors, also uses the network element. If an object initially was programmed in FBD or LD and then is converted to IL, the networks will be still present in the IL program. Vice versa, if you start programming an object in IL, you need at least 1 network element which might contain all instructions, but you can also use further networks to structure the program.

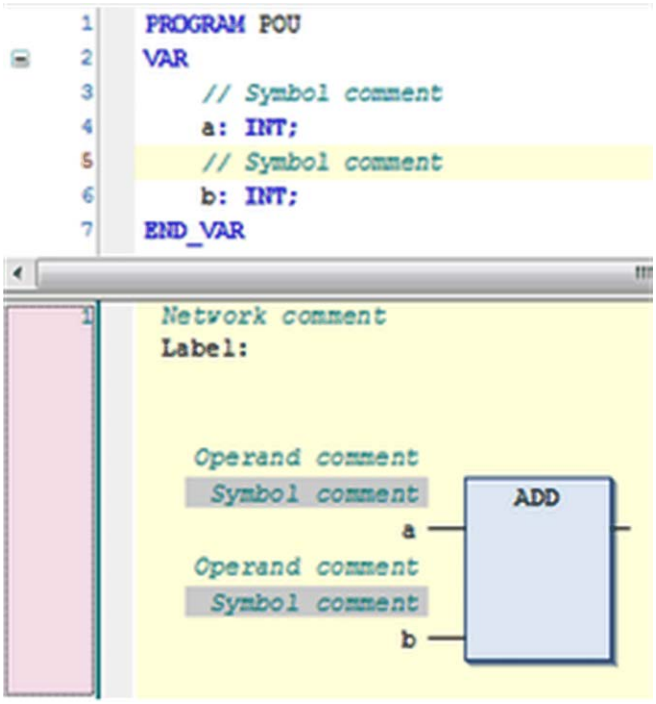
A network optionally can get assigned a title, a comment and a label, page 294.

You can switch the visibility of the title, the comment fields, and the network separator on and off in the FBD, LD and IL editor options dialog box. If the option is activated, click in the network directly below the upper border to open an edit field for the title. For entering a comment, correspondingly open an edit field directly below the title field. The comment can be multi-lined. Press ENTER to insert line breaks. Press CTRL + ENTER to terminate the input of the comment text.

Whether and how a network comment is displayed in the editor, is defined in the FBD, LD, and IL editor options dialog box.

To add a label, page 294, which then can be addressed by a jump, page 294, use the command **Insert label** . If a label is defined, it will be displayed below the title and comment field or - if those are not available - directly below the upper border of the network.

Comments and label in a network



You can set a network in comment state. This indicates that the network is not processed but displayed and handled as a comment. To achieve this, use the command **Toggle network comment state**.

On a currently selected network (cursor position 6, page 284), you can apply the default commands for copying, cutting, inserting, and deleting.

> **NOTE:** Right-clicking (cursor position 6, page 284) titles, comments, or labels will select this entry only instead of the whole network. So the execution of the default commands does not affect the network.

To insert a network, use command **Insert Network** or drag it from the toolbox, page 291. A network with all belonging elements can also be copied or moved, page 274 by drag and drop within the editor.

You can also create subnetworks, page 295 by inserting branches.

RET Network

In online mode, an additional empty network will be displayed below the existing networks. Instead of a network number, it is identified by RET.

It represents the position at which the execution will return to the calling POU and provides a possible halt position, page 287.

Assignment in FBD/LD/IL

Overview

Depending on the selected cursor position, page 283 in FBD or LD, an assignment will be inserted directly in front of the selected input (cursor position 2,

page 284), directly after the selected output (cursor position 4, page 284) or at the end of the network (cursor position 6, page 284). In an LD network, an assignment will be displayed as a coil, page 302. Alternatively, drag the assignment element from the toolbox, page 291 or copy or move, page 274 it by drag and drop within the editor view.

After insertion, the text string `???` can be replaced by the name of the variable that is to be assigned. For this, use the **...** button to open the **Input Assistant**.

In IL, page 269, an assignment is programmed via LD and ST instructions. Refer to *Modifiers and Operators in IL*, page 270.

Jump in FBD/LD/IL

Overview

Depending on the selected cursor position, page 283 in FBD, page 268 or LD, page 269, a jump will be inserted directly in front of the selected input (cursor position 2), directly after the selected output (cursor position 4) or at the end of the network (cursor position 6). Alternatively, drag the jump element from the toolbox, page 291 or copy or move, page 274 it by drag and drop within the editor.

After insertion, you can replace the automatically entered `???` by the label to which the jump should be assigned.

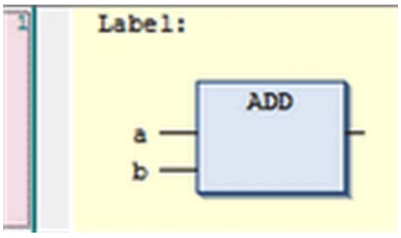
In IL, page 269, a jump is inserted via a JMP instruction. See in this context the description of the operators and modifiers in IL, page 270.

Label in FBD/LD/IL

Overview

Below the network comment field each FBD, page 268, LD, page 269 or IL network have a text input field for defining a label. The label is an optional identifier for the network and can be addressed when defining a jump, page 294. It can consist of any sequence of characters.

Position of a label in a network



See the **Tools > Options > FBD, LD and IL editor** dialog box for defining the display of comment and title.

Boxes in FBD/LD/IL

Overview

A box, insertable in an FBD, page 268, LD, page 269, or IL, page 269 network, is a complex element and can represent additional functions like timers, counters, arithmetic operations, or also programs, IEC functions and IEC function blocks.

A box can have one or more inputs and outputs and can be provided by a library or can be programmed by the user.

You can group the inputs and outputs of a function block in its declaration via the `attribute pingroup`. In the display of the function block in an FBD or LD editor, the groups can then be faded out and in. For further information, refer to the *Attribute pingroup* chapter, page 625.

If provided with the respective module and if the option **Show box icon** is activated, an icon is displayed within the box.

Use in FBD, LD

You can position a box in a LD network or in an FBD network by using the command **Insert Box**, **Insert Empty Box**. Alternatively, you can insert it from the toolbox, page 291 or copy or move it within the editor via drag and drop. For further information, refer to the description of the **Insert Box** command.

Use in IL

In an IL, page 269 program, a CAL, page 270 instruction with parameters will be inserted in order to represent a box element.

You can update the box parameters (inputs, outputs) - in case the box interface has been modified - with the current implementation without having to reinsert the box by executing the **Update parameters** command.

`RETURN` Instruction in FBD/LD/IL

Overview

With a `RETURN` instruction, the FBD, page 268, LD, page 269 or IL, page 269 POU can be exited.

In an FBD or LD network, you can place it in parallel or at the end of the previous elements. If the input of a `RETURN` is TRUE, the processing of the POU will immediately be exited.

Execute the command **Insert Return** to insert a `RETURN` instruction. Alternatively, drag the element from the toolbox, page 291 or copy or move, page 274 it from another position within the editor.

`RETURN` element



In IL, the RET, page 270 instruction is used for the same purpose.

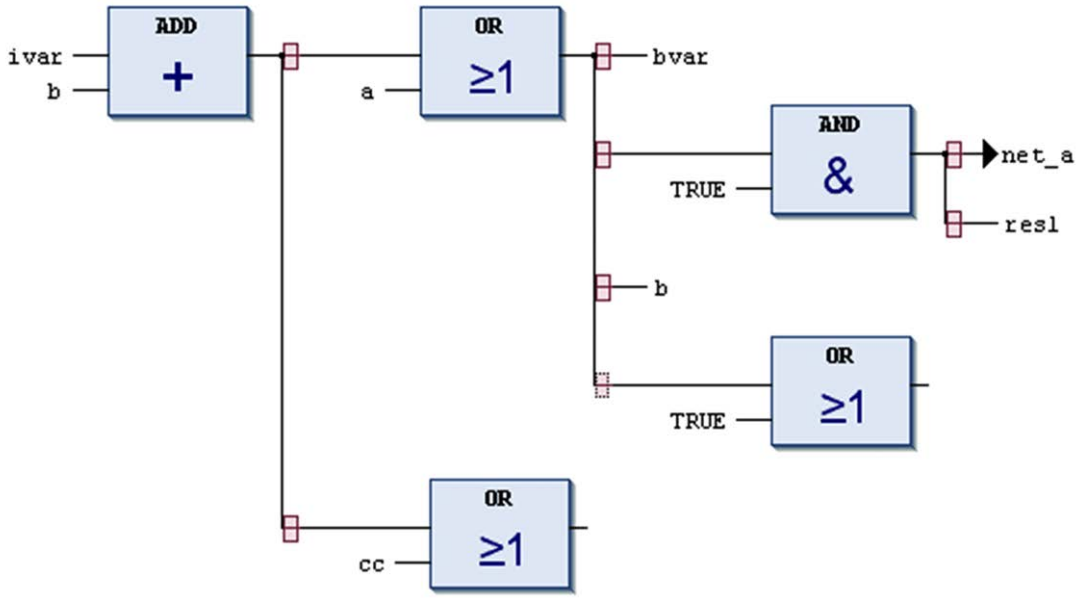
Branch / Hanging Coil in FBD/LD/IL

Overview

In a Function Block Diagram, page 268 or Ladder Diagram, page 269 network, a branch or a hanging coil splits up the processing line as from the current cursor position. The processing line will continue in 2 subnetworks which will be executed 1 after each other from top to bottom. Each subnetwork can get a further branch, such allowing multiple branching within a network.

Each subnetwork gets an own marker (an upright rectangle symbol). You can select it (cursor position 11, page 284) in order to perform actions on this arm of the branch.

Branch markers

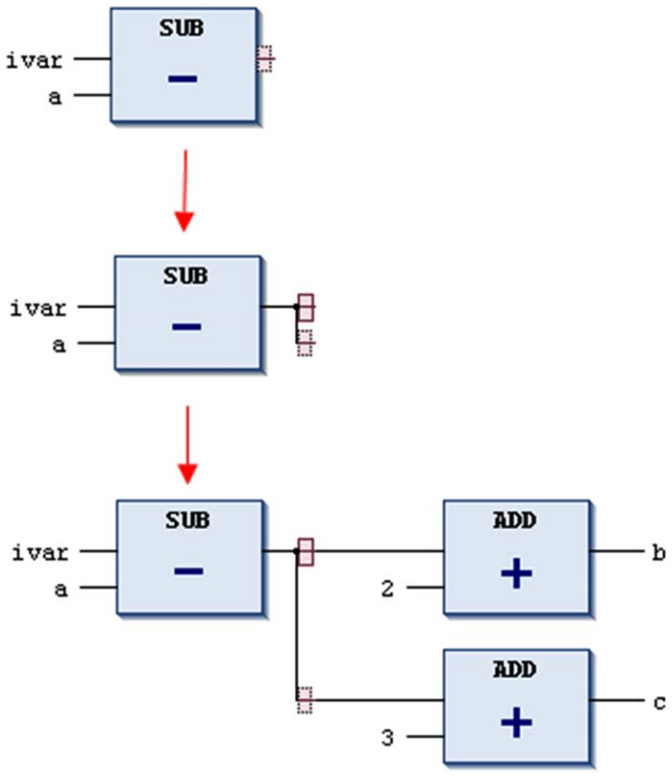


In FBD, insert a branch via command **Insert branch**. Alternatively, drag the element from the toolbox, page 291. For the possible insert positions, refer to the description of the **Insert branch** command.

NOTE: Cut and paste is not implemented for subnetworks.

A branch has been inserted at the *SUB* box output in the example shown below. This created 2 subnetworks, each selectable by their subnet marker. After that, an *ADD* box was added in each subnetwork.

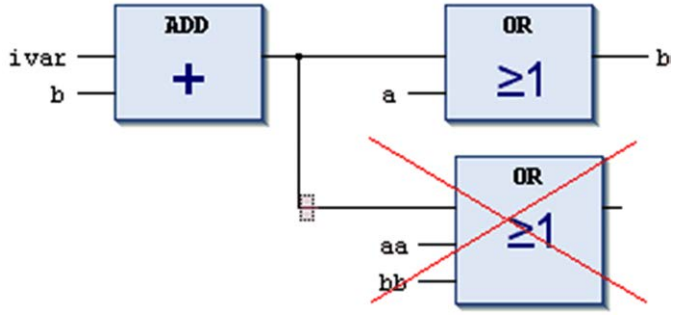
Network in FBD with inserted branch



To delete a subnetwork, first remove all elements of the subnetwork, that is all elements which are positioned to the right of the marker of the subnetwork. Then select the marker and execute the standard **Delete** command or press the **Delete** key.

In the following image, the 3-input-OR element has to be deleted before you can select and delete the marker of the lower subnetwork.

Delete branch or subnetwork



Execution in Online Mode

The particular branches will be executed from left to right and then from top to bottom.

IL (Instruction List)

The IL, page 269 does not support networks with branches. They will stay in the original representation.

Parallel Branches

You can use parallel branches for setting up parallel branch, page 297 evaluation in ladder networks.

In contrast to the open branch (without the junction point), the parallel branches are closed. They have common split and junction points.

Parallel Branch

Overview

A parallel branch allows you to implement a parallel evaluation of logical elements. This is accomplished via a methodology described as Short Circuit Evaluation (SCE). SCE allows you to by-pass the execution of a function block with a boolean output if certain parallel conditions are evaluated to be TRUE. The condition can be represented in the LD editor by a parallel branch to the function block branch. The SCE condition is defined by 1 or several contacts within this branch, connected parallel or sequentially.

The vertical connections of short circuit evaluation branches executed in parallel are represented by a double line in order to differentiate them from OR constructs that are represented by a single line (see the figure *Parallel branch for SCE in a ladder network*).

> **NOTE:** The term branch is also used for another element that splits off a signal flow. This branch, page 295 as opposed to the parallel branch has no junction point.

The parallel branch works as follows: first it will be parsed for the branches not containing a function block. If 1 of such branches is evaluated to be TRUE, then the function block in the parallel branch will not be called and the value at the input of the function block branch will be passed to the output. If the SCE condition is evaluated to be FALSE, then the function block will be called and the boolean result of the function block execution call will be passed on.

If all branches contain function blocks, then they will be evaluated in top-to-bottom order and the outputs of them will be combined with logical OR operations. If there

are no branches containing a function block call, then the normal OR operation will be performed.

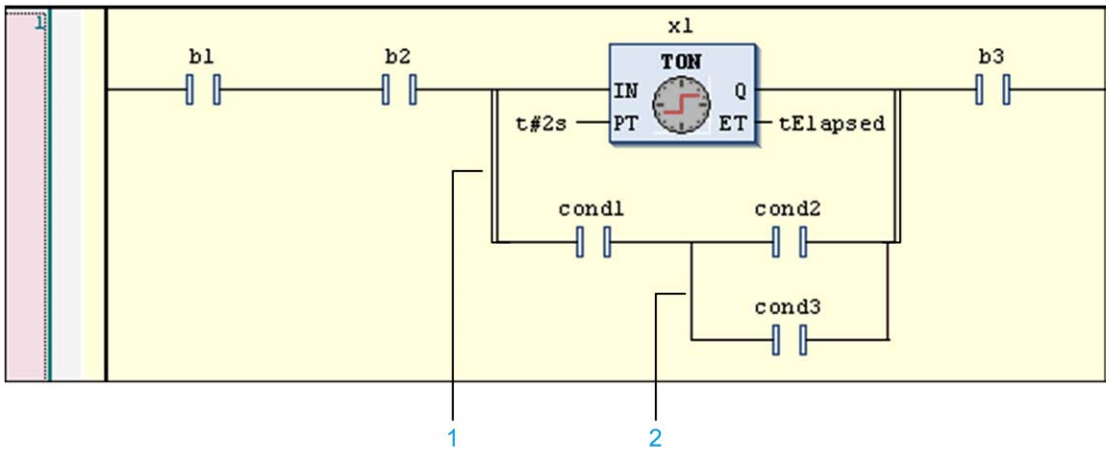
To insert a parallel branch with SCE function, select the function block box and execute the command **Insert Contact Parallel above** or **Insert Contact Parallel below**. This is only possible if the first input and the main output of the function block are of type BOOL.

Below is an example of the generated language model for the given network.

Example for SCE

The function block instance *x1 (TON)* has a boolean input and a boolean output. Its execution can be skipped if the condition in the parallel branch is evaluated to be TRUE. This condition value results from the *OR* and *AND* operations connecting the contacts *cond1*, *cond2* and *cond3*.

Parallel branch for SCE in a ladder network



1 The double vertical connection lines indicate a construct that is subject to an SCE.

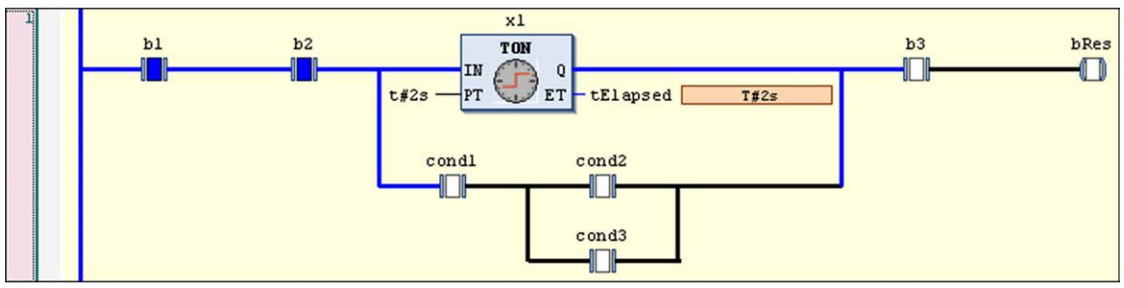
2 The single vertical connection line indicates an OR construct.

The processing is as shown in the following, whereby *P_IN* and *P_OUT* represent the boolean value at the input (split point) and output (junction point) of the parallel branch, respectively.

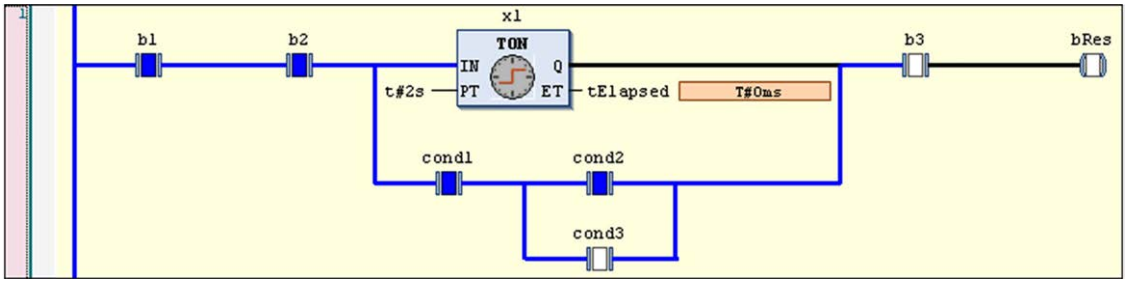
```
P_IN := b1 AND b2;
IF ((P_IN AND cond1) AND (cond2 OR cond3)) THEN
   P_OUT := P_IN;
ELSE
   x1(IN := P_IN, PT := {p 10}t#2s);
   tElapsed := x1.ET;
   P_OUT := x1.Q;
END_IF
bRes := P_OUT AND b3;
```

The following images show the dataflow (blue) in case the function block is executed (condition resulting from *cond1*, *cond2* and *cond3* is FALSE) or bypassed (condition is TRUE).

Condition=FALSE, function block is executed:



Condition=TRUE, function block is bypassed:



Branch Start / End

Overview

The branch start / end element is an LD element.

It allows you to define the start / end point of a branch. If there is no start point defined, you can use the element to define it. Is the start point already defined the element determines the end point.

For further information, refer to the chapter *Parallel Branch*, page 297.

Set/Reset in FBD/LD/IL

FBD and LD

A boolean output in FBD, page 268 or correspondingly an LD, page 269 coil can be set or reset. To change between the set states, use the respective command **Set/Reset** from the contextual menu when the output is selected. The output or coil will be marked by an *S* or an *R*.

Set	If value TRUE arrives at a set output or coil, this output/coil will become TRUE and remain TRUE. This value cannot be overwritten at this position as long as the application is running.
Reset	If value TRUE arrives at a reset output or coil, this output/coil will become FALSE and remain FALSE. This value cannot be overwritten at this position as long as the application is running.

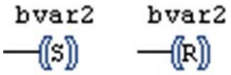
Set output in FBD



In the LD editor, you can insert set and reset coils by drag and drop. To perform this action, use either the **ToolBox**, category **Ladder elements**, or the `S` and `R` elements from the tool bar.

Example:

Set coil, reset coil



For further information, see Set/Reset Coil, page 300.

IL

In an Instruction List, use the S and R, page 271 operators to set or reset an operand.

Set/Reset Coil

Overview

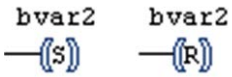
Coils, page 302 can also be defined as set or reset coils.

You can recognize a set coil by the `S` in the coil symbol: *(S)*. A set coil will not overwrite the value TRUE in the appropriate boolean variable. That is, the variable once set to TRUE remains TRUE.

You can recognize a reset coil by the `R` in the coil symbol: *(R)*. A reset coil will not overwrite the value FALSE in the appropriate boolean variable. That is, the variable once set to FALSE will remain FALSE.

In the LD editor, you can insert set coils and reset coils directly via drag and drop from the **ToolBox**, category **Ladder elements**. In doing so, you can also replace already inserted coil elements by others.

Set coil, reset coil



Execute

Overview

The `Execute` element is a block with EN/ENO that you can insert in an FBD or LD network and that you can fill with ST code. The ST code is executed when the block is activated for processing with a TRUE signal at the EN input.

Drag the `Execute` element from the toolbox, page 291 into the network or execute the **FBD/LD/IL > Execute** command (see Menu Commands, Online Help) to insert the element.

Entering ST Code

Step	Action	Comment
1	Click the input field with the text `Enter ST-Code here...` in the box.	The ST editor, page 356 opens providing the usual functionalities.
2	Enter ST code.	The necessary variable declarations are inserted in the declaration section of the FBD/LD/IL function block.
3	Complete entering code by clicking out of the block or pressing **Ctrl** + **Enter**.	–

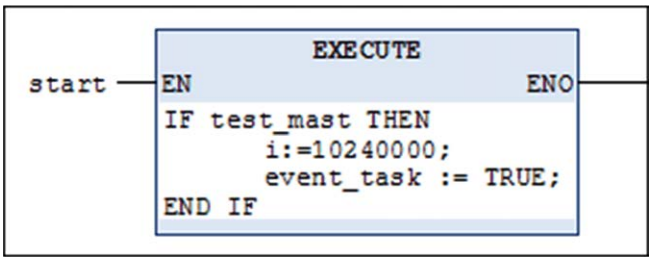
Online Mode

In online mode, you can open the ST editor by clicking the plus sign under the `EN` input on the function block. The usual online functionalities (monitoring, debugging) are available in the editor.

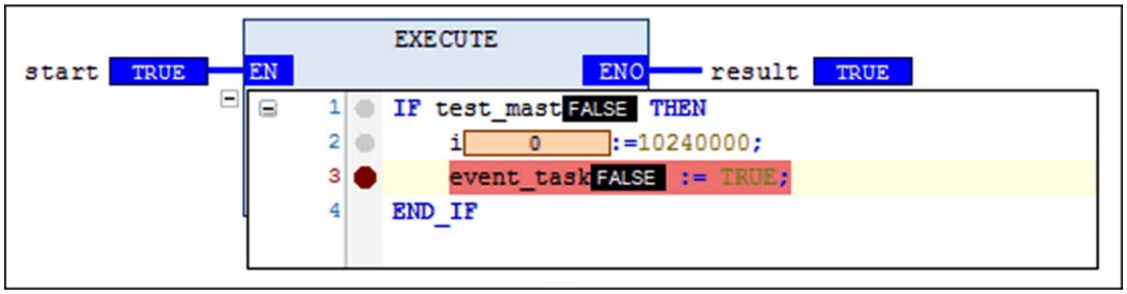
Example

Examples for an `Execute` block in the FBD network.

Offline mode, ST program inserted



Online mode, ST editor open



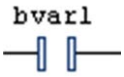
LD Elements

Contact

Overview

This is an LD element.

In LD, page 269 in its left part, each network contains 1 or several contacts. Contacts are represented by 2 vertical, parallel lines.

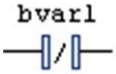


Contacts pass on condition ON (TRUE) or OFF (FALSE) from left to right. A boolean variable is assigned to each contact. If this variable is TRUE, the condition is passed from left to right and finally to a coil in the right part of the network. Otherwise the right connection receives the value FALSE.

You can connect multiple contacts in series as well as in parallel. Contacts in parallel represent a logical 'OR' condition such that only one of them need be TRUE to have the parallel branch transmit the value TRUE. Conversely, contacts in series represent a logical 'AND' condition whereas all the contacts must be TRUE to have the final contact transmit TRUE.

Therefore, the contact arrangement corresponds to either an electric parallel or a series circuit.

A contact can also be negated. This is indicated by the slash in the contact symbol.



A negated contact passes on the incoming condition (TRUE or FALSE) only if the assigned boolean variable is FALSE.

You can insert a contact in an LD network via one of the commands **Insert Contact** or **Insert Contact (right) Insert Contact Parallel (above)**, **Insert Contact Parallel (below)**, **Insert Rising Edge Contact**, or **Insert Falling Edge Contact** which are part of the **LD** menu. Alternatively, you can insert the element via drag and drop from the **ToolBox**, page 291 or from another position within the editor (drag and drop).

You can replace an already inserted contact by a new contact or a negated contact. For this purpose, drag a contact or a negated contact from the toolbox, page 291 onto an existing contact and drop it there.

FBD and IL

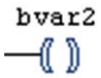
If you are working in FBD, page 268 or IL, page 269 view, the command will not be available. But contacts and coils inserted in an LD network will be represented by corresponding FBD elements or IL instructions.

Coil

Overview

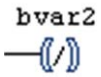
This is an LD element.

On the right side of an LD network, there can be any number of coils which are represented by parentheses.



They can only be arranged in parallel. A coil transmits the value of the connections from left to right and copies it to an appropriate boolean variable. At the entry line, the value ON (TRUE) or the value OFF (FALSE) can be present.

Coils can also be negated. This is indicated by the slash in the coil symbol.



In this case the negated value of the incoming signal will be copied to the appropriate boolean variable.

You can insert a coil in a network via one of the commands **Insert Coil**, **Insert Set Coil**, **Insert Reset Coil**, or **Insert Negated Coil** in the **LD** menu. Alternatively, you can insert the element via drag and drop from the **ToolBox** (**Ladder elements**) or via drag and drop from another position within the editor. Also refer to *Set and Reset Coils*, page 300.

FBD and IL

If you are working in FBD, page 268 or IL, page 269 view, the command will not be available. But contacts and coils inserted in an LD network will be represented by corresponding FBD elements or IL instructions.

Ladder Editor

What's in This Chapter

General Information on the Ladder Editor 303
Programming in the Ladder Editor 304
Monitoring and Detecting Errors 307
Navigating in the Ladder Editor 309

General Information on the Ladder Editor

Overview

The Ladder editor is a network-based editor for the IEC programming language LD (Ladder Diagram). Compared to the FBD/LD/IL Editor, page 268, it provides simplified editing capabilities. In the FBD/LD/IL editor, it was required to insert separate elements from the **ToolBox** for variants of elements. With the Ladder editor, variants of elements are created according to the selected insertion position or by switching a modifier.

To convert function blocks that were created in the FBD/LD/IL editor to the Ladder format, use the **Convert to New Ladder** command (refer to the Menu Commands Online Help).

How to Create Network Elements

To program a circuit diagram in the implementation part of the Ladder editor, create one or more network elements as follows:

Step	Action
1	Insert the required elements from the **ToolBox** or by using the Ladder editor commands (refer to the Menu Commands Online Help).
2	Link the elements.
3	Assign inputs, outputs and modifiers to the elements.

Optionally, assign the following elements to a network element:

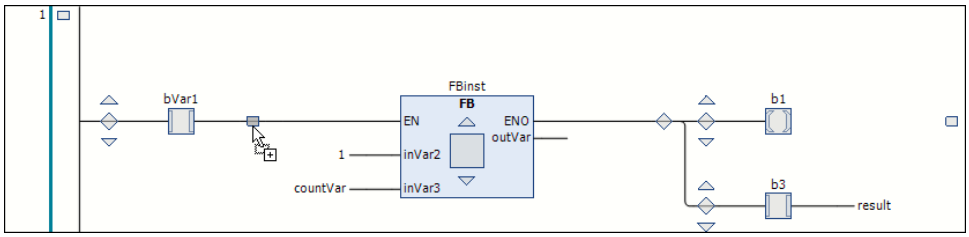
- Title
- Comment
- Jump label

It is also possible to comment out a network.

How to Insert Elements

To insert an element, drag it from the **ToolBox** to a possible insertion position that is indicated when you move the element over the implementation part. Insertion positions are indicated by the following symbols:

- Square with a gray background inside an existing element symbol
- Rhombus on a connecting line
- Triangle pointing up or down for insertion above or below



While moving the mouse over the implementation part, possible positions are indicated by a plus symbol that is displayed at the cursor. When you release the mouse button, the element is inserted. Selected areas in the editor are highlighted in red and outlined. Commands for modifying (**Edge Detection**, **Set/Reset**, **EN/ENO**), deleting, refactoring, searching are available in the contextual menu.

Online Mode and General Settings

In online mode, you can perform monitoring and debugging with breakpoints as well as writing and forcing of values.

Configure settings concerning the editing in the Ladder editor in the **Tools > Options > Ladder Editor** dialog box (refer to the Menu Commands Online Help).

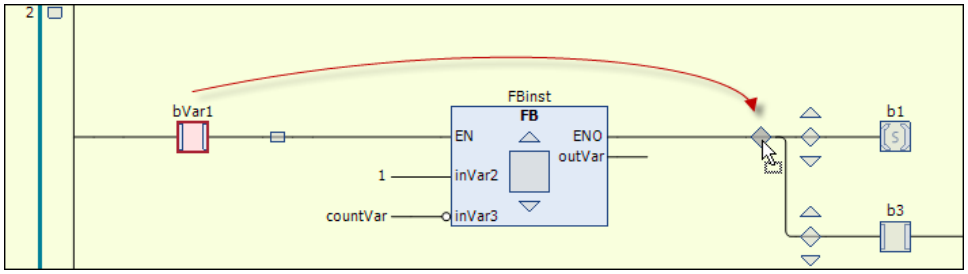
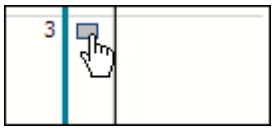
Programming in the Ladder Editor

Overview

This section lists instructions that describe individual actions when working in the Ladder editor. A specific program is not created.

As a prerequisite, a POU that is created in the Ladder Diagram implementation language is available in your project. This POU must be open and editable in the Ladder editor.

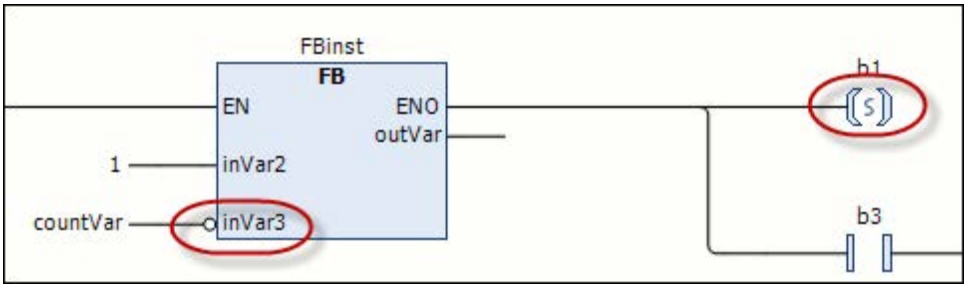
Inserting, Updating, Repositioning, and Modifying Elements in a Network

Task	Description
Inserting an element	Insert an element as described in How to Insert Elements, page 304. The element is inserted in the processing line of the network.
Inserting a block	Insert a **Block** element in the network. Double-click the three question marks [???] in the element box to replace them with the name of the block from the project or a library. It is a good practice to use the **Input Assistant** by clicking the **...** button.
Updating a block	After you have called a function block from your project in the ladder editor and modified the function block, run the **Ladder > Update Parameters** command (refer to the Menu Commands Online Help) to update in the ladder diagram.
Inserting an input	Insert an **Input** element before the **Block**. To assign variables, replace the three question marks [???] of the element with the variable name. The **Auto Declare** dialog box opens, if configured (refer to the Menu Commands Online help).
Selecting / repositioning elements	To reposition an element, select it in the network so that it is highlighted in red. If, for example, a block is selected, the box inside the block symbol is displayed with a red background. Drag the element to a possible insertion position that is indicated by the cursor symbol becoming a square, and release the mouse button. 
Replacing elements	To replace an existing element with another one, drag the new element onto the one you want to replace. When the element to be replaced becomes green, release the mouse button.
Adding a new network	To add a new network, drag the **Network** element to the implementation part. As a result, square insertion points are displayed in the left margin area where the network numbering is displayed. The new network is inserted above the insertion point. 

Assigning Modifiers

To assign modifiers to elements, select, for example, the input pin of a function block and open the contextual menu. Select the **Negate** command to negate the input, or one of the **Set/Reset** or edge detection commands (**Edge detection > Rising Edge**, **Edge detection > Falling Edge**). As a result, the input is marked with the corresponding symbol.

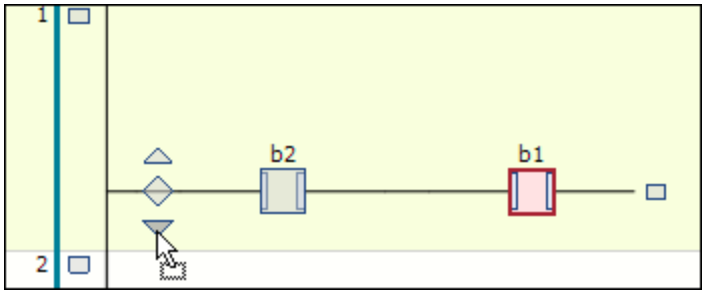
Example: Negated input at the block, set coil



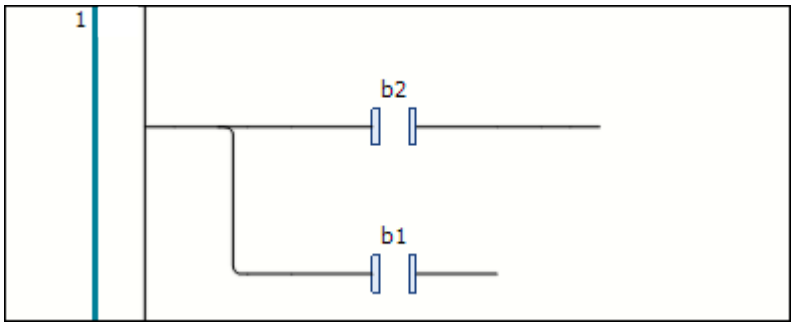
Creating and Editing Branches

In contrast to the FBD/LD/IL Editor, page 268, the Ladder editor does not provide elements for **Branches**. Create and edit branches by positioning or repositioning elements as follows.

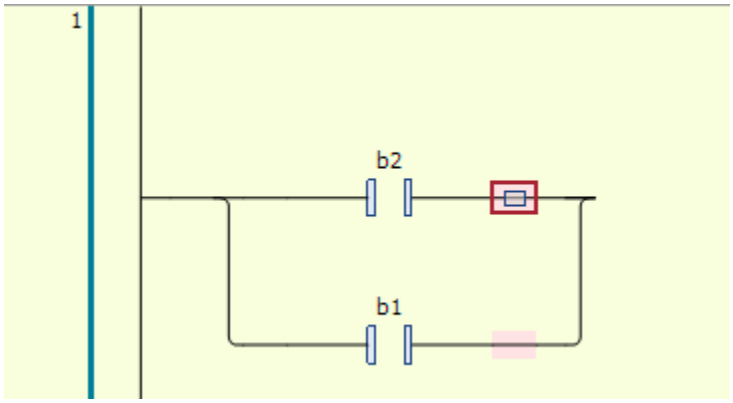
1. Create the following network:



2. Drag the second contact to the insertion position marked with a downward pointing arrow in front of the first contact. As a result, a parallel branch open to the end is created, in which each branch contains a contact.



3. To create a closed branch from the open branch (to program an OR construct), select both branches after the contact element (multiselection). The selection is indicated by the small square with a red background on the line. Right-click and run the **Close Parallel Branch** command. As a result, the two parallel branches that were open at the end are converted to a closed branch.



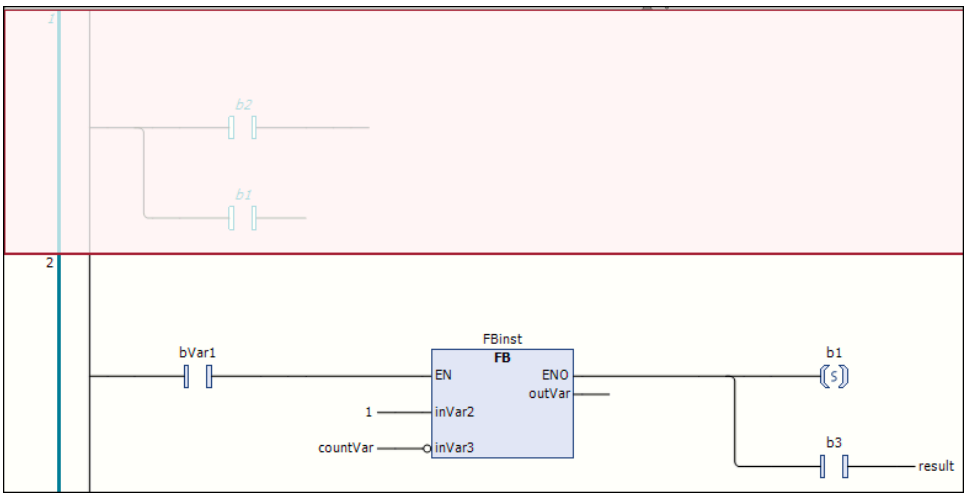
There are two options to reopen a closed branch:

- Select one of the two branches and drag the selection box to that of the other branch.
- Select both branches and run the **Open Parallel Branch** command.

  NOTE: If you have closed branches of multiple parallel branches, then the **Open Parallel Branch** command opens the branches.

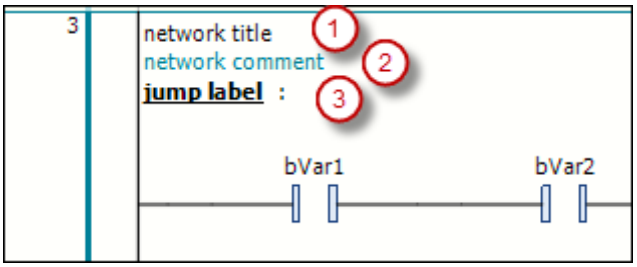
Commenting Out a Network

To comment out a network, select a network so that it is displayed with a red background. From the contextual menu, run the **Outcommented** command. As a result, the network contents are displayed in gray and the texts in italics. The network is not included in the executable object file.



Adding a Network Title, Network Comment or Jump Label

You can add a network title (1), a network comment (2), and/or a jump label (3) to a network. To achieve this, click in the first, second, or third line in the upper left corner of the **Network** element. A jump label is used as a target when inserting a **Jump** element.



Refer to the **Tools > Options > Ladder Editor** dialog box (in the Menu Commands Online Help) to display the elements.

Monitoring and Detecting Errors

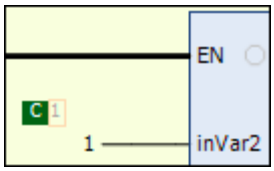
Overview

In online mode, the Ladder editor supports value monitoring and the writing and forcing of values. You can set breakpoints, and the color-coded representation of the connections allows for a flow control with calculated values.

Monitoring

In online mode, the value of each variable is displayed in the editor. Constant variables are indicated by a green C symbol. Configure the display of the values in the **Tools > Options > Ladder Editor** dialog box (refer to the Menu Commands Online Help).

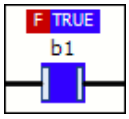
Example of a constant value:



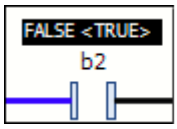
Writing and Forcing Values

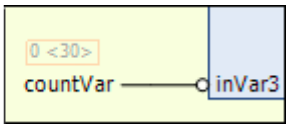
A variable that is being forced, is indicated by a red F symbol before the forced value. If a value has been prepared for writing or forcing, then this value is displayed directly after the present value in angle brackets.

Example of a forced value:



Example of a prepared value:





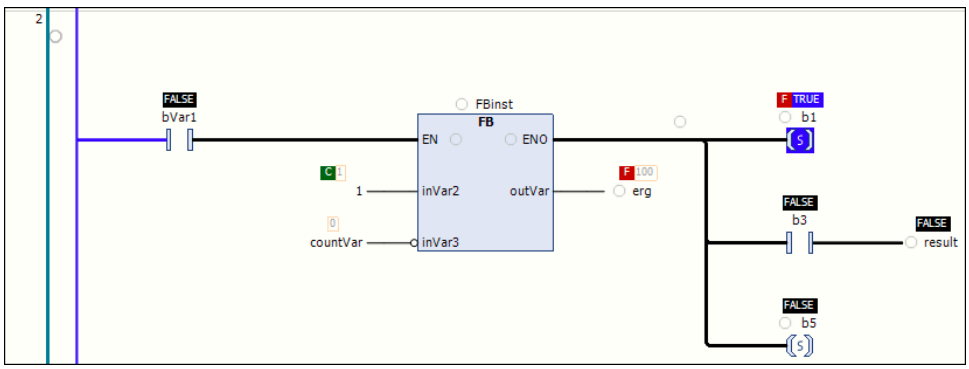
Color-Coded Representations of Connections

In the online view of a Ladder diagram, the connecting lines are displayed in different colors.

- Connections with the value TRUE are indicated by a thick blue line.
- Connections with the value FALSE are indicated by a thick black line.
- Connections with an undetermined or analog value are displayed as a thin black line.

  NOTE: The value of the connections is not read from the monitored variables, but it is calculated by the programming system. Thus, this is not a genuine flow control.

Example of connecting lines and breakpoint positions:



Breakpoints

Breakpoints can be set at positions where the values of variables can change (instructions), where the program branches, or where another POU is called. Possible breakpoint positions are indicated by an empty gray circle. Set breakpoints are displayed as a solid red circle. Refer to the figure indicating connecting lines and breakpoint positions.

Breakpoints can be set at the following positions:

- On a POU that can be called (function block, function, program, action, method). It is not possible with operator blocks (for example: `ADD`, `DIV`).
- On assignments.
- Before parallel branches.
- At the end of the block at the position of the return to the calling block.
- On `EN` input and `ENO` output of a block.
- On the network, this indicates that a breakpoint is set in the network.

NOTE: A breakpoint is automatically set in the different methods that can be called. Thus, if a method managed by an interface is called, then breakpoints are set in the methods of function blocks that implement this interface as well as in the derived function blocks that use the method. If a method is called by a pointer to a function block, then the breakpoints are set in the method of the function block and in the derived function blocks that use the method.

Navigating in the Ladder Editor

The following keys and key combinations switch between cursor positions in the editor. Switching is also possible across networks.

Key or key combination	Effect
→ ←	Switch to the adjacent cursor position, along the signal flow (from left to right, and right to left).
↑↓	Switch to the next cursor position above or below the present position if this adjacent position belongs to the same logical group. Example: The connections of a block form a logical group. If such a logical group does not exist, then change to the first cursor position in the next higher or lower adjacent element. In the case of parallel-connected elements, navigation is performed along the first branch.
Ctrl + **Home**	Switch to the first network and select it.
Ctrl + **End**	Switch to the last network and select it.
Page Up ↑	Scroll up one page. The top network on this page is selected.
Page Down ↓	Scroll down one page. The bottom network on this page is selected.

Continuous Function Chart (CFC) Editor

What's in This Chapter

Continuous Function Chart (CFC) Language 310
CFC Editor 310
Cursor Positions in CFC 312
CFC Elements / ToolBox 314
Working in the CFC Editor 319
Execution Order in CFC 322
CFC Editor in Online Mode 325
CFC Editor Page-Oriented 328

Continuous Function Chart (CFC) Language

Overview

The Continuous Function Chart is an extension to the IEC 61131-3 standard, and is a graphical programming language based on the Function Block Diagram (FBD) language, page 268. However, in contrast to the FBD language, there are no networks. CFC allows the free positioning of graphic elements, which in turn allows for feedback loops.

For creating CFC programming objects in EcoStruxure Machine Expert, see the description of the CFC editor, page 310.

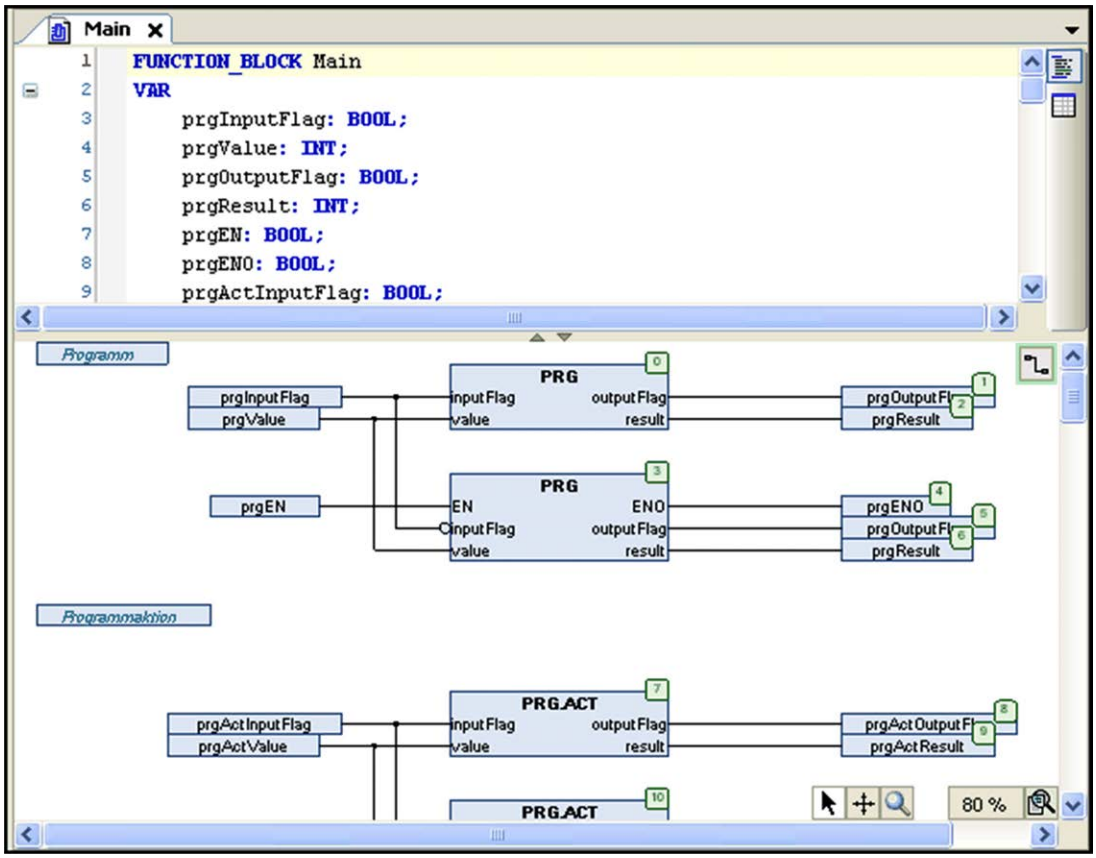
CFC Editor

Overview

The CFC editor is a graphical editor available for programming objects in the continuous function chart (CFC) programming language , page 310, which is an extension to the IEC 61131-3 programming languages. Choose the language when you add a new program organization unit (POU) object to your project. For large projects, consider using the page-oriented version, page 328.

The editor will be available in the lower part of the window which opens when opening a CFC POU object. This window also includes the declaration editor, page 373 in its upper part.

CFC editor



The CFC editor in contrast to the FBD / LD editor allows free positioning, page 319 of the elements, which allows direct insertion of feedback paths. The sequence of processing is determined by a list which contains all currently inserted elements and can be modified.

The following elements are available in a toolbox, page 314 and can be inserted via drag and drop:

- box (operators, functions, function blocks, and programs)
- input
- output
- jump
- label
- return
- composer
- selector
- connection marks
- comments

You can connect the input and output pins of the elements by drawing a line with the mouse. The path of the connecting line will be created automatically and will follow the shortest possible route. The connecting lines are automatically adjusted as soon as the elements are moved. For further information, refer to the description of inserting and arranging elements, page 319. For complex charts, you can use connection marks, page 315 instead of lines. You may also consider the possibility of modifying the routing.

It may happen that elements get positioned in a way that they cover already routed connections. These collisions are indicated by red connection lines. If there are any collisions in the chart, the button in the upper right corner of the editor view gets a red outline: . To edit the collisions step by step, click this button and execute the command **Show next collision**. Then the next found concerned connection will be selected.

For complex charts, you can use connection marks, page 315 instead of lines. You may also wish to use the page-oriented version of the CFC editor.

A zoom function allows you to change the dimension of the editor window: Use the  button in the lower right corner of the window and choose between the listed zoom factors. Alternatively, you can select the entry **...** to open a dialog box where you can type in any arbitrary factor.

You can call the commands for working in the CFC editor from the contextual menu or from the **CFC** menu which is available as soon as the CFC editor is active.

Cursor Positions in CFC

Overview

Cursor positions in a CFC program are indicated by a gray background when hovering over the programming element.

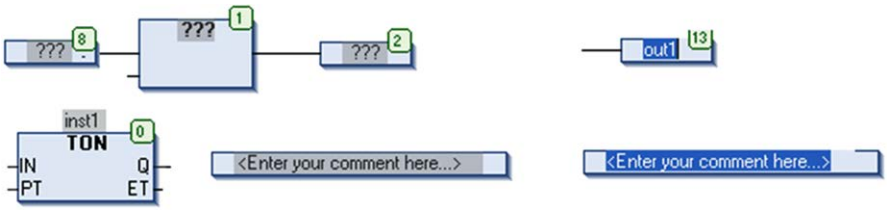
When you click one of these shadowed areas, before releasing the mouse-button, the background color will change to red. As soon as you release the mouse-button, this will become the current cursor position, with the respective element or text being selected and displayed as red-colored.

There are 3 categories of cursor positions. See the possible positions indicated by a gray shaded area as shown in the illustrations of the following paragraphs.

Cursor Positioned on a Text

If the cursor is positioned on a text and you click on the mouse-button, it is displayed as blue-shaded and can be edited. The **...** button is available for opening the input assistant. Primarily after having inserted an element, the characters ??? represent the name of the element. Replace these characters by a valid identifier. After that a tool tip is displayed by positioning the cursor on the name of a variable or a box parameter. The tool tip contains the type of the variable or parameter and, if it exists, the associated comment in a second line.

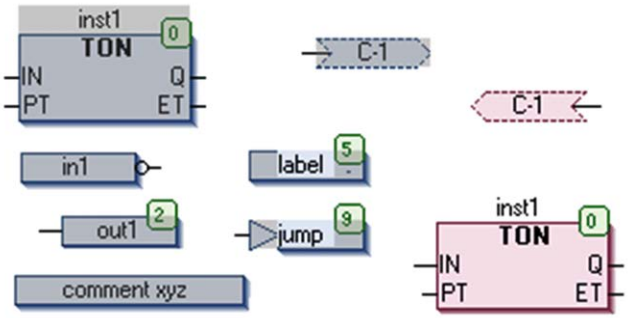
Possible cursor positions and an example of selected text:



Cursor Positioned on the Body of an Element

If the cursor is positioned on the body of an element (box, input, output, jump, label, return, comment, connection mark), these will be displayed as red-colored and can be moved by moving the mouse.

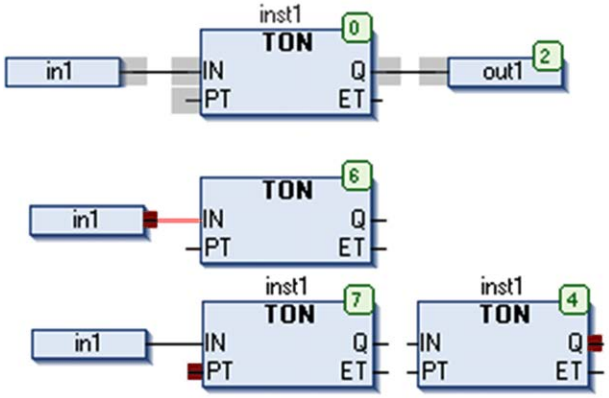
Possible cursor positions and example of a selected body:



Cursor Positioned on the Body on Input or Output Connection of an Element

If the cursor is positioned on an input or output connection of an element, a red filled square will indicate that position (point of connection). It can be negated or set/reset.

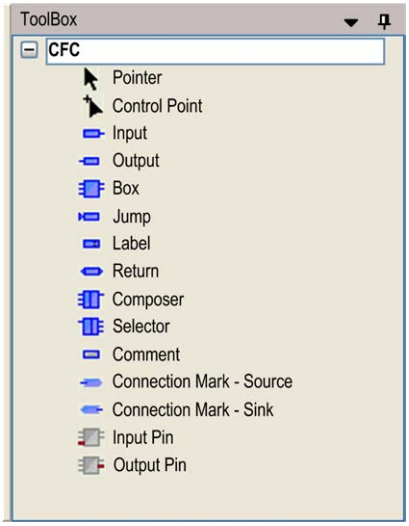
Possible cursor positions (gray shadows) and examples of selected output and input positions (red squares):



CFC Elements / ToolBox

Overview

The graphical elements available for programming in the CFC editor, page 310 window are provided by a toolbox. Open the toolbox in a view window by executing the command **ToolBox** in the **View** menu.

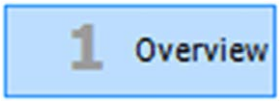
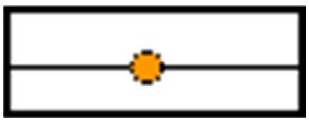


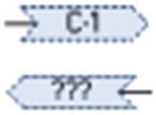
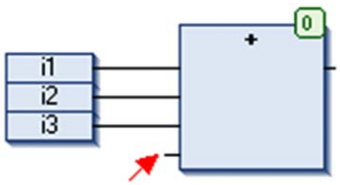
Select the desired element in the toolbox and insert, page 319 it in the editor window via drag and drop.

Besides the programming elements, there is an entry 

, at the top of the toolbox list. As long as this entry is selected, the cursor has the shape of an arrow and you can select elements in the editor window for positioning and editing.

CFC Elements

Name	Symbol	Description
page		The number of the page is given automatically according to its position. You can enter the name (`Overview` in this example) in the orange field at the top of the page.
control point		A control point is needed to fix a manually modified connection line routing. This helps to prevent the modification from being reverted by the command **Route all Connections**. By 2 control points you can mark a definite segment of a line for which you want to modify the routing.
input		You can select the text offered by `???` and replace it by a variable or constant. The input assistance serves to select a valid identifier.
output		
box		You can use a box to represent operators, functions, function blocks, and programs. You can select the text offered by `???` and replace it by an operator, function, function block, or program name. The input assistance serves to select one of the available objects. If you insert a function block, another `???` will be displayed above the box. Replace the question marks by the name of the function block instance. If a function block with constant input parameters is instantiated, then the box element shows a field **Parameters...** in the bottom left corner of the box. Click this button to open a dialog box for editing the input parameters. Refer to the *Edit Parameters...* chapter (see Menu Commands, Online Help). If you replace an existing box by another (by modifying the entered name) and the new one has a different minimum or maximum number of input or output pins, the pins will be adapted correspondingly. If pins are to be removed, the lowest one will be removed first.
jump		Use the jump element to indicate at which position the execution of the program should continue. This position is defined by a label (see below). Therefore, replace the text offered by `???` by the label name.
label		A label marks the position to which the program can jump (see the element jump). In online mode, a return label for marking the end of POU is automatically inserted.
return		In online mode, a return element is automatically inserted in the first column and after the last element in the editor. In stepping, it is automatically jumped to before execution leaves the POU.
composer		Use a composer to handle an input of a box which is of type of a structure. The composer will display the structure components and thus make them accessible in the CFC for the programmer. For this purpose name the composer like the concerned structure (by replacing `???` by the name) and connect it to the box instead of using an input element.
selector		A selector in contrast to the composer is used to handle an output of a box which is a type of structure. The selector will display the structure components and thus make them accessible in the CFC for the programmer. For this purpose, name the selector like the concerned structure (replace `???` by the name) and connect it to the box instead of using an output element.
comment		Use this element to add any comments to the chart. Select the placeholder text and replace it with any desired text. To obtain a new line within the comment, press CTRL + ENTER.

Name	Symbol	Description
connection mark – source connection mark – sink		You can use connection marks instead of a connection line, page 320 between elements. This can help to clear complex charts. For a valid connection, assign a connection mark – source element at the output of an element and assign a connection mark – sink (see below) at the input of another element. Assign the same name to both marks (no case-sensitivity). Naming: The first connection mark – source element inserted in a CFC by default is named `C-1` and can be modified manually. In its counterpart connection mark – sink, replace the `???` by the same name string as used in the source mark.The editor will verify that the names of the marks are unique. If the name of a source mark is changed, the name of the connected sink mark will automatically be renamed as well. However, if a sink mark is renamed, the source mark will keep the old name. This allows you to reconfigure connections. Likewise, removing a connection mark does not remove its counterpart. To use a connection mark in the chart, drag it from the toolbox to the editor window and then connect its pin with the output or input pin of the respective element. Alternatively you can convert an existing normal connection line by using the command **Connection Mark**. This command allows you to change connection marks back to normal connection lines as well. For figures showing some examples of connection marks, refer to the chapter *Connection Mark*.
input pin		Depending on the box type, you can add an additional input. For this purpose, select the box element in the CFC network and draw the input pin element on the box. You can drag an input or output connection to another position at the box while keeping pressed the **Ctrl** key.
output pin	–	Depending on the box type, you can add an additional output. For this purpose, select the box element in the CFC network and draw the output pin element on the box. You can drag an input or output connection to another position at the box while keeping pressed the **Ctrl** key.

Example of a Composer

A CFC program `cfc_prog` handles an instance of function block `fublo1`, which has an input variable `struvar` of type structure. Use the composer element to access the structure components.

Structure definition `stru1` :

```
TYPE stru1 :
STRUCT
  ivar:INT;
  strvar:STRING:='hallo';
END_STRUCT
END_TYPE
```

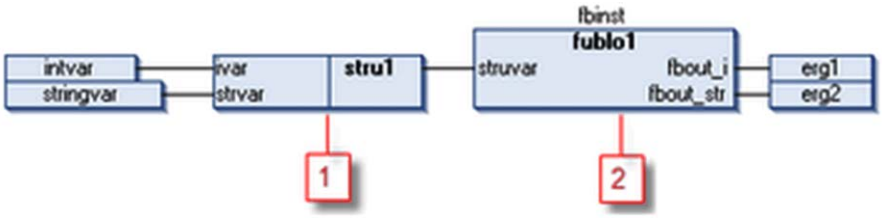
Declaration and implementation of function block `fublo1`:

```
FUNCTION_BLOCK fublo1
VAR_INPUT
  struvar:STRU1;
END_VAR
VAR_OUTPUT
  fbout_i:INT;
  fbout_str:STRING;
END_VAR
VAR
  fbvar:STRING:='world';
END_VAR
fbout_i:=struvar.ivar+2;
fbout_str:=CONCAT (struvar.strvar,fbvar);
```

Declaration and implementation of program `cfc_prog`:

```
PROGRAM cfc_prog
VAR
  intvar: INT;
  stringvar: STRING;
  fbinst: fublo1;
  erg1: INT;
  erg2: STRING;
END_VAR
```

Composer element



1 composer

2 function block with input variable `struvar` of type structure `stru1`

Example of a Selector

A CFC program `cfc_prog` handles an instance of function block `fublo2`, which has an output variable `fbout` of type structure `stru1`. Use the selector element to access the structure components.

Structure definition `stru1`:

```
TYPE stru1 :
STRUCT
  ivar:INT;
  strvar:STRING:='hallo';
END_STRUCT
END_TYPE
```

Declaration and implementation of function block `fublo1`:

```
FUNCTION_BLOCK fublo2
VAR_INPUT CONSTANT
  fbin1:INT;
  fbin2:DWORD:=24354333;
  fbin3:STRING:='hallo';
END_VAR
VAR_INPUT
  fbin : INT;
END_VAR
VAR_OUTPUT
  fbout : stru1;
  fbout2:DWORD;
END_VAR
VAR
  fbvar:INT;
  fbvar2:STRING;
END_VAR
```

Declaration and implementation of program `cfc_prog`:

```
VAR
  intvar: INT;
  stringvar: STRING;
  fbinst: fublo1;
  erg1: INT;
  erg2: STRING;
  fbinst2: fublo2;
END_VAR
```

The illustration shows a selector element where the unused pins have been removed by executing the command **Remove Unused Pins**.



1 function block with output variable `fbout` of type structure `stru1`

2 selector

Working in the CFC Editor

Overview

The elements available for programming in the CFC editor are provided in the **ToolBox**, page 314 which by default is available in a window as soon as the CFC editor is opened.

The **Tools > Options > CFC editor** defines general settings for working within the editor.

Inserting

To insert an element, select it in the **ToolBox**, page 314 by a mouse-click, keep the mouse-button pressed and drag the element to the desired position in the editor window. During dragging, the cursor will be displayed as an arrow plus a rectangle and a plus-sign. When you release the mouse-button, the element will be inserted.

Selecting

To select an inserted element for further actions such as editing or rearranging, click an element body to select the element. It will be displayed by default as red-shaded. By additionally pressing the SHIFT key, you can click and select further elements. You can also press the left mouse-button and draw a dotted rectangle around all elements which you want to select. As soon as you release the button the selection will be indicated. By command **Select all** , all elements are selected at once.

By using the arrow keys you can shift the selection mark to the next possible cursor position. The sequence depends on the execution order or the elements, which is indicated by element numbers, page 321.

When an input pin is selected and you press CTRL + LEFT ARROW, the corresponding output will be selected. When an output pin is selected and you press CTRL + LEFT ARROW, the corresponding outputs will be selected.

Drag and Drop of Variables

You can directly insert variables from the GVL, page 205 or a POU, page 173 as an input or output element of a function block. To achieve this, select the variable in the GVL or POU, and drag it to the input or output pin of a function block. The input or output element for this variable is automatically created and connected to the pin of the function block.

Replacing Boxes

To replace an existing box element, replace the currently inserted identifier by that of the desired new element. The number of input and output pins will be adapted if necessary due to the definition of the POUs and thus some existing assignments could be removed.

Moving

To move an element, select the element by clicking the element body (see possible cursor positions, page 312) and drag it, while keeping the mouse-button

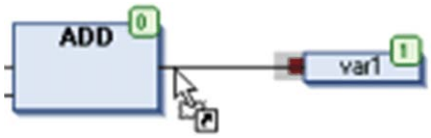
pressed, to the desired position. Then release the mouse-button to place the element. You also can use the **Cut** and **Paste** commands for this purpose.

Connecting

You can connect the input and output pins of 2 elements by a connection line or via connection marks.

Connection line: You can either select a valid point of connection that is an input or output pin of an element (refer to *Cursor Positions in CFC*, page 312), and then draw a line to another point of connection with the mouse. Or you can select 2 points of connection and execute the command **Select connected pins**. A selected possible point of connection is indicated by a red filled square. When you draw a line from such a point to the target element, you can identify the possible target point of connection. When you then position the cursor over a valid connection point, an arrow symbol is added to the cursor when moving over that point, indicating the possible connection.

The following figure provides an example: After a mouse-click on the input pin of the `var1` element, the red rectangle is displayed showing that this is a selected connection point. By keeping the mouse button pressed, move the cursor to the output pin of the ADD box until the cursor symbol appears as shown in the figure. Now release the mouse button to establish the connection line.



The shortest possible connection is created taking into account the other elements and connections.

You can drag an input or output connection to another position at the box while keeping pressed the **Ctrl** key.

Connection marks: you could as well use connection marks instead of connection lines in order to simplify complex charts. Refer to the description of connection marks, page 315.

Copying

To copy an element, select it and use the **Copy** and **Paste** commands.

Editing

After you have inserted an element, by default the text part is represented by `???`. To replace this by the desired text (POU name, label name, instance name, comment, and so on), click the text to obtain an edit field. Also the button **...** will be available to open the **Input Assistant**.

Deleting

You can delete a selected element or connection line by executing the command **Delete**, which is available in the contextual menu or press the **Delete** key.

Opening a Function Block

If a function block is added to the editor, you can open this block with a double-click. Alternatively, use the command **Browse > Go To Definition** from the contextual menu.

Execution Order, Element Numbers

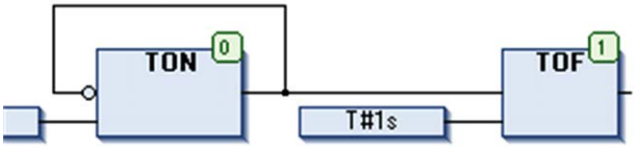
The sequence in which the elements in a CFC network are executed in online mode is indicated by numbers in the upper right corner of the box, output, jump, return, and label elements. The processing starts at the element with the lowest number, which is 0.

You can modify the execution order by commands which are available in the submenu **Execution Order** of the **CFC** menu.

When adding an element, the number will automatically be given according to the topological sequence (from left to right and from top to bottom). The new element receives the number of its topological successor if the sequence has already been changed, and all higher numbers are increased by 1.

Consider that the number of an element remains constant when it is moved.

Consider that the sequence influences the result and must be changed in certain cases.



Changing Size of the Working Sheet

In order to get more space around an existing CFC chart in the editor window, you can change the size of the working area (working sheet). Do this by selecting and dragging all elements with the mouse or use the cut-and-paste commands (refer to *Moving*, page 319)

Alternatively, you can use a special dimension settings dialog box. This may save time in the case of large charts. Refer to the description of the **Edit Working Sheet** dialog box (see Menu Commands, Online Help). In case of page-oriented CFC, you can use the **Edit Page Size** command (see Menu Commands, Online Help).

Execution Order in CFC

Overview

The execution order of elements within POUs is uniquely determined in text-based and network-based editors. In the CFC editor, however, you can position the elements freely, so the execution order is initially not unique. Therefore, EcoStruxure Machine Expert determines the execution order by data flow and, in the case of multiple networks, by the topological position of the elements. The elements are sorted from top to bottom and from left to right to achieve a unique execution order.

The chronological order of the elements in the chart can be indicated by temporarily displaying the execution order. For networks with feedback, you can define an element as the starting point in the feedback loop.

EcoStruxure Machine Expert V2.0 and later versions allow you to edit the execution order in a CFC object explicitly by selecting the **Explicit Execution Order Mode** in the **CFC Execution Order** tab of the CFC object (see Menu Commands, Online Help) **Properties**.

Data Flow

In general, the term data flow describes the chronological order of reading or writing which data when and how in which programming object. A POU can process an arbitrary number of data flows. These data flows can also be executed independently of each other.

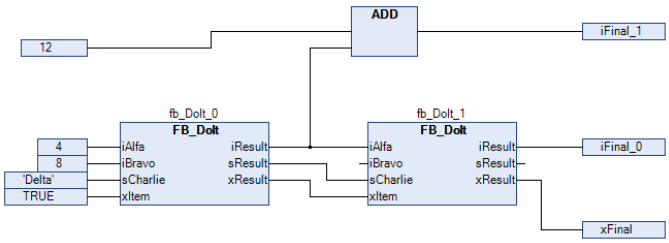
Displaying the Execution Order

By default, the **Auto Data Flow Mode** is selected in the **CFC Execution Order** tab of the CFC object (see Menu Commands, Online Help) **Properties** and the execution order of a CFC object is determined automatically.

To temporarily display the execution order in the CFC editor, proceed as follows:

Step	Action	Comment
1	Create a new project using the **Standard project** template and specify a name.	Example name: `Minimal`

2	Insert the function block `FB_DOIt` with inputs and outputs in ST implementation language.	Example:
		`FUNCTION_BLOCK FB_DoIt`
		`VAR_INPUT`
		`iAlfa : INT;`
		`iBravo: INT;`
		`sCharlie : STRING := 'Charlie';`
		`xItem : BOOL;`
		`END_VAR`
		`VAR_OUTPUT`
		`iResult : INT;`
		`sResult : STRING;`
		`xResult : BOOL;`
		`END_VAR`
		`VAR`
		`END_VAR`
		`iResult := iAlfa + iBravo;`
		`IF xItem = TRUE THEN`
		`xResult := TRUE;`
		`END_IF`

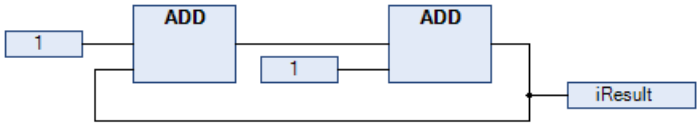
3	Create the function block *ExecuteCFC* in CFC implementation language.	Example:
		`PROGRAM ExecuteCFC`
		`VAR`
		`fb_DoIt_0: FB_DoIt;`
		`fb_DoIt_1: FB_DoIt;`
		`iFinal_1: INT;`
		`iFinal_0: INT;`
		`xFinal: BOOL;`
		`END_VAR`
		
		
		Programming objects recently created in CFC have selected the option **Auto Data Flow Mode**. The optimum execution order of the programming objects is defined internally.

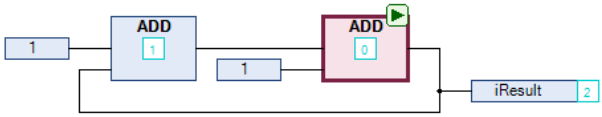
4	Execute the command **CFC > Execution Order > Display Execution Order**.	**Result**: The execution order of the object is displayed: The boxes and inputs are numbered according to the chronological processing order. This temporary display is removed as soon as you click again in the CFC editor.

Manually Determining the Execution Order in Feedback Networks

To manually determine the execution order in feedback networks, proceed as follows:

Step	Action	Comment

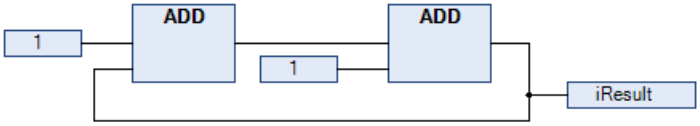
1	Create a CFC program with feedback.	Example: The POU `PrgPositiveFeedback` counts.
		`PROGRAM PrgPositiveFeedback`
		`VAR`
		`  iResult: INT;`
		`END_VAR`
		

2	Select an element within the feedback.	**Result**: The selected element is highlighted red.
3	Execute the command **CFC > Execution Order > Set Start of Feedback**.	**Result**: The selected element is assigned number `0` (the lowest number of the feedback) and is indicated by the  symbol. At runtime, this POU is processed first. 

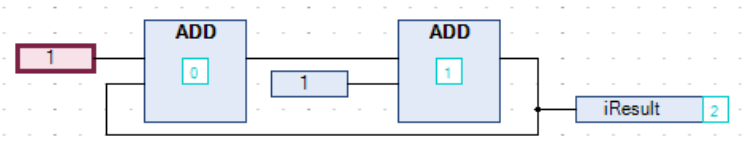
To undo this numbering, proceed as follows:

1. Select the POU defined as start POU.
2. Execute the command **CFC > Execution Order > Set Start of Feedback**.

Result: The POU is no longer defined as start POU and the execution order is defined internally.



To display the execution order by data flow, execute the command **CFC > Execution Order > Display Execution Order**.



Manually Defining the Execution Order

The default **Auto Data Flow Mode** determines the order of CFC objects automatically to achieve execution that is optimized by time and by cycle. With this mode activated, you do not need to care for defining the execution order during the development process.

NOTE: The **Explicit Execution Order Mode** allows you to define the execution order manually. Note, that it is your responsibility to adapt the execution order and to assess the consequences and impacts. For your support, the execution order is permanently displayed.

To manually define the execution order, proceed as follows:

Step	Action	Result
1	In the **Devices** or **POUs** tree, right-click a CFC object and execute the command **Properties**.	The **Properties** dialog box opens.
2	Select the **CFC Execution Order** tab.	The **Execution order** list displays the selected mode.
3	From the **Execution order** list, select the option **Explicit Execution Order Mode**.	–
4	Click **OK** to confirm and to close the dialog box.	• The **Explicit Execution Order Mode** is activated. • The networks are numbered in the CFC editor. • The following commands are available in the **CFC > Execution Order** menu: ◦ **Display Execution Order** command (see Menu Commands, Online Help) ◦ **Set Start of Feedback** command (see Menu Commands, Online Help)
5	Open a CFC object.	–
6	Select a numbered element and execute the command **CFC > Execution Order > Send to Front**.	The execution order is resorted and the selected element is assigned the number 0.

CFC Editor in Online Mode

Overview

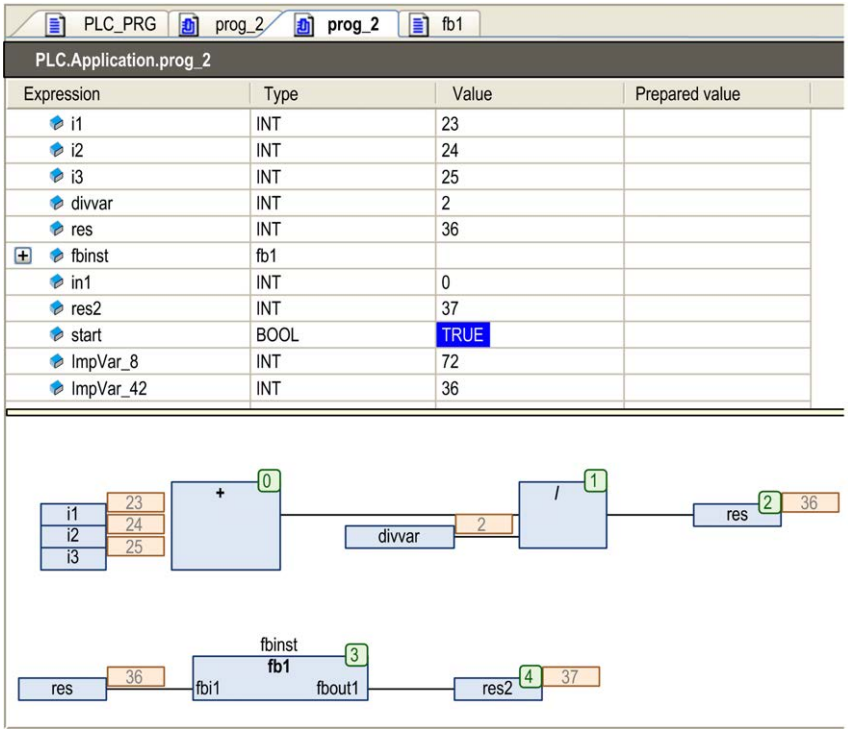
In online mode, the CFC editor provides views for monitoring, writing and forcing the variables and expressions on the controller. The debugging functionality (breakpoints, stepping, and so on) is available as described below.

- For information on how to open objects in online mode, refer to the description of the user interface in online mode, page 50.
- The editor window of a CFC object also includes the declaration editor in the upper part. Refer to the description of the declaration editor in online mode, page 378.
- For information on how to edit parameters in the CFC editor, refer to the description of the **Edit Parameters...** command (see Menu Commands, Online Help).

Monitoring

The actual values are displayed in small monitoring windows behind each variable (inline monitoring).

Online view of a program object PLC_PRG:



Note for the online view of a function block POU, that monitoring is only possible in an instance view. In the base implementation of the function block POU, no values are displayed. The column **Value** contains the text **Value of the expression** and the inline monitoring fields in the implementation part show three question marks each.

Boolean connections are monitored in the colors TRUE = blue and FALSE = black.

Forcing/Writing Variables

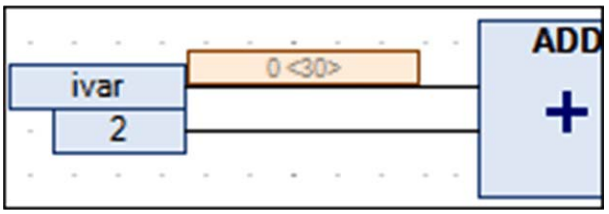
In online mode, you can prepare a value for forcing or writing a monitored variable either in the declaration editor or - if the option **Prepare values in implementation part** is activated - also in the implementation part. For working in the declaration editor, refer to the chapter *Declaration Editor in Online Mode*, page 378. In the implementation part, click the monitoring box next to the respective element or directly the element to open the **Prepare Value** dialog box, page 358. For boolean variables, no dialog box opens, but clicking the value currently displayed next to the variable directly toggles between the possible values to be forced or written. In the monitoring box of a currently forced variable a red **F** is displayed.

Forced value in the implementation part



If the CFC option **Prepare values in implementation part** is enabled, then the value currently prepared for writing or forcing is displayed behind the current value in angle brackets in the monitoring field of a variable.

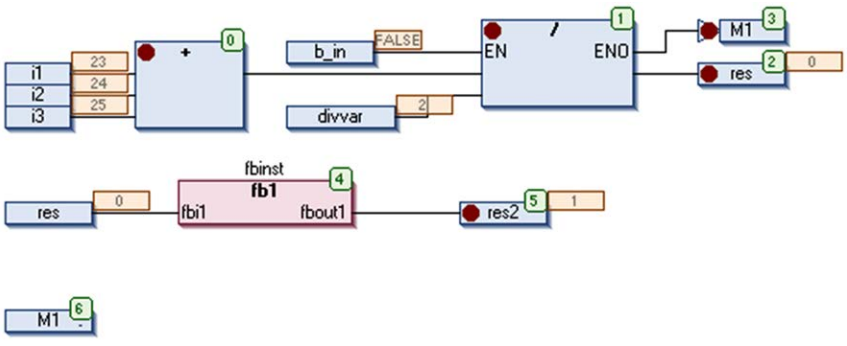
Prepared value in the implementation part



Breakpoint Positions in CFC Editor

The possible breakpoint positions basically are those positions in a POU at which values of variables can change or at which the program flow branches out or another POU is called. See possible positions in the following image.

Breakpoint positions in CFC editor:



NOTE: A breakpoint will be set automatically in all methods which may be called. If an interface-managed method is called, breakpoints will be set in all methods of function blocks implementing that interface and in all derivative function blocks subscribing the method. If a method is called via a pointer on a function block, breakpoints will be set in the method of the function block and in all derivative function blocks which are subscribing to the method.

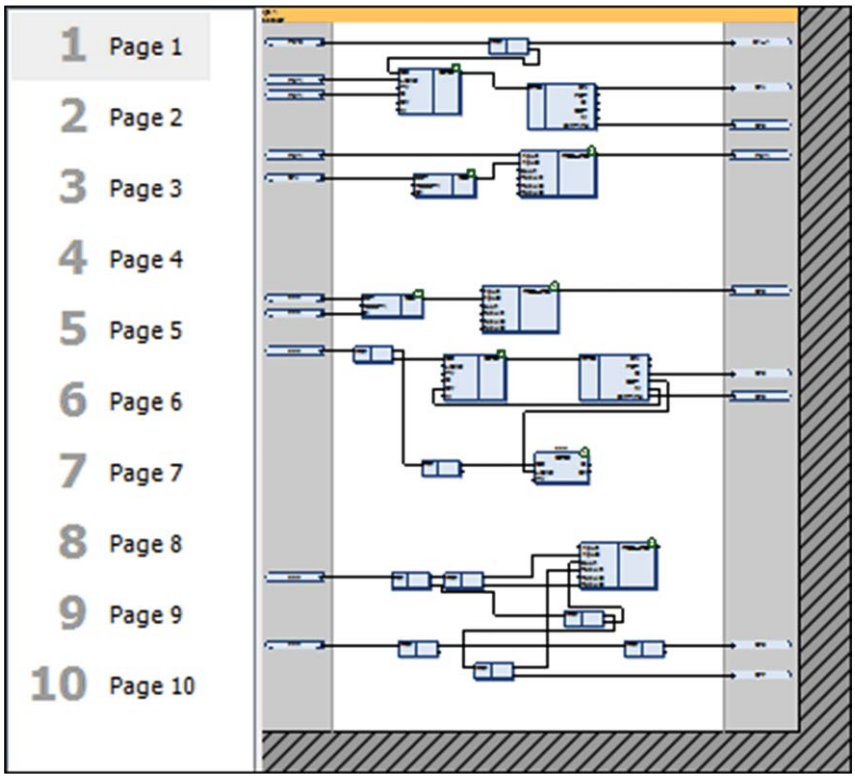
CFC Editor Page-Oriented

Overview

In addition to the CFC standard editor, EcoStruxure Machine Expert provides the CFC editor pagination. Besides the tools, page 314 and commands of the standard CFC editor, this editor allows you to arrange the elements on any number of different pages.

NOTE: You cannot convert POUs created in the language CFC page-oriented to normal CFC and vice versa. You can copy elements between these 2 editors with the copy and paste commands (via clipboard) or the drag and drop function.

CFC pagination



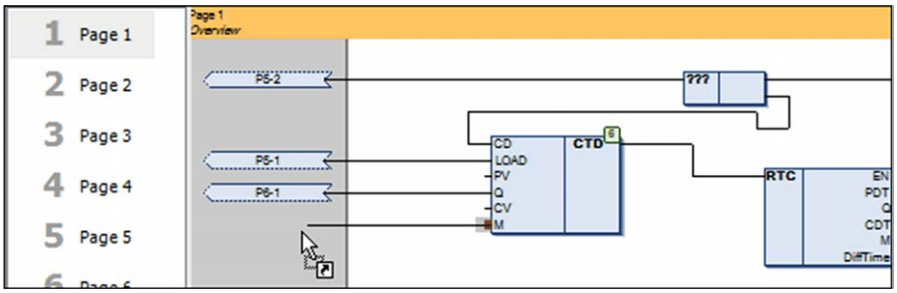
To change the size of the page execute the command **Edit Page Size**.

Connections Between 2 Pages

Connections between 2 pages are realized with the elements **connection mark – source** and **connection mark – sink** (refer to the description of connection marks). You can place the **connection mark – source** by drag and drop to the right margin - the **connection mark – sink** to the left margin. If you draw a connection line from an input or output of an element to the margin, the connection mark is placed automatically. Click the **...** button to open the list of connection marks provided by the **Input Assistant**.

Use the arrow keys to navigate within the diagram from one element to the next.

Insertion of connection marks



Execution Order

The execution order of the pages is from top to the bottom. Within a page, the order follows the rules of the standard CFC editor (refer to further information of execution order, element numbers, page 321 and to the chapter *Execution Order in CFC*, page 322). You can change the execution order of elements only within the associated page. You cannot change the execution order of elements on different pages.

Sequential Function Chart (SFC) Editor

What's in This Chapter

SFC Editor 330
SFC - Sequential Function Chart Language 331
Cursor Positions in SFC 332
Working in the SFC Editor 333
SFC Element Properties 335
SFC Elements / ToolBox 336
Qualifier for Actions in SFC 346
Implicit Variables - SFC Flags 347
Sequence of Processing in SFC 353
SFC Editor in Online Mode 354

SFC Editor

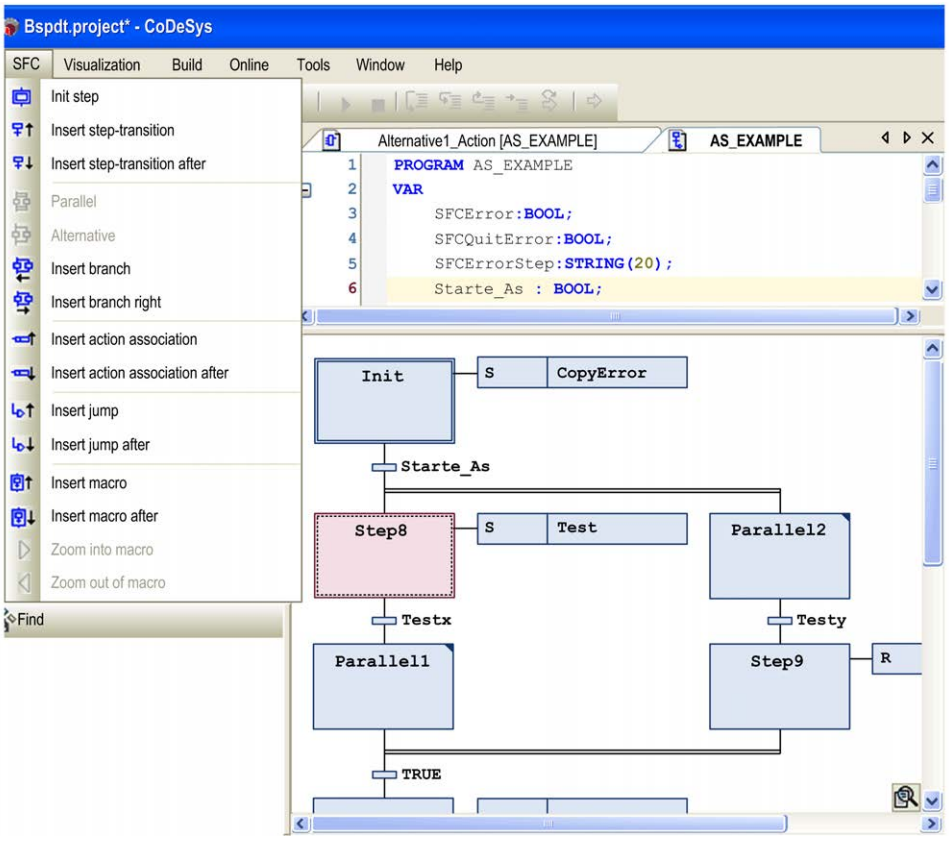
Overview

The SFC editor is available for programming objects in the IEC 61131-3 programming language SFC - Sequential Function Chart, page 331. Choose the language when you add a new POU object to the project.

The SFC editor is a graphical editor. Perform general settings concerning behavior and display in the **Options > SFC editor** dialog box.

The SFC editor is available in the lower part of the window which opens when you edit an SFC POU object. This window also includes the **Declaration Editor**, page 373 in the upper part.

SFC editor



Working with the SFC Editor

The elements, page 336 used in an SFC diagram are available in the **SFC** menu. The menu is available as soon as the SFC editor is active. Arrange them in a sequence or in parallel sequences of steps which are connected by transitions. For further information, refer to *Working in the SFC Editor*, page 333.

You can edit the properties of steps in a separate properties, page 335 window. Among others, you can define the minimum and maximum time of activity for each step.

You can access implicit variables, page 347 for controlling the processing of an SFC (for example, step status, timeout analyzation, reset).

SFC - Sequential Function Chart Language

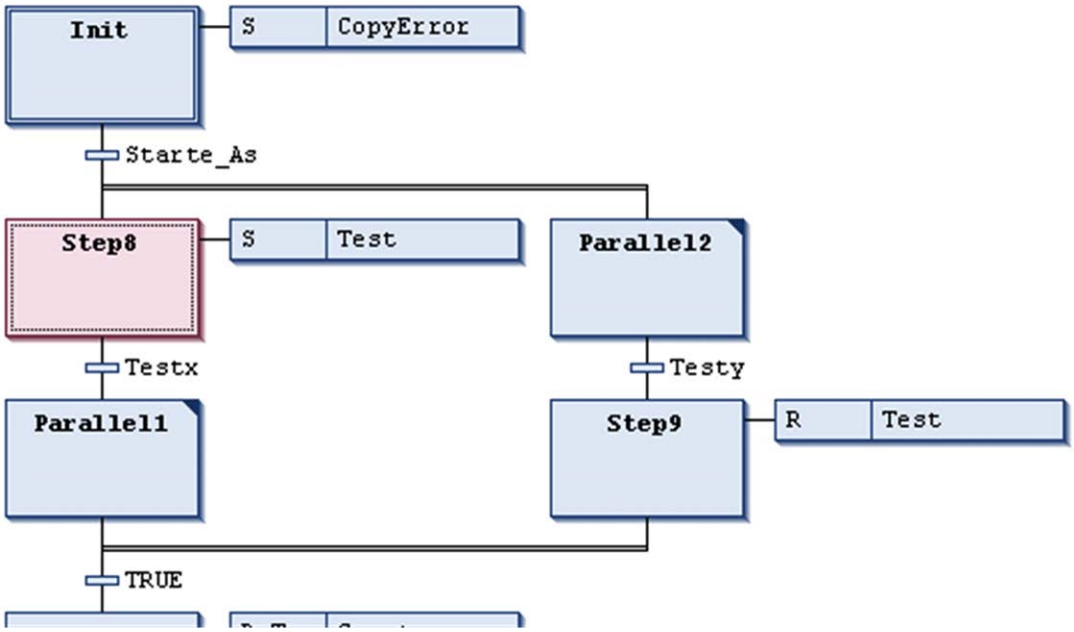
Overview

The Sequential Function Chart (SFC) is a graphically oriented language which describes the chronological order of particular actions within a program. These actions are available as separate programming objects, written in any available programming language. In SFC, they are assigned to step elements and the sequence of processing is controlled by transition elements. For a detailed description on how the steps will be processed in online mode, refer to *Sequence of Processing in SFC*, page 353.

For information on how to use the SFC editor in EcoStruxure Machine Expert, refer to the description of the *SFC Editor*, page 330.

Example

Example for a sequence of steps in an SFC module:



Cursor Positions in SFC

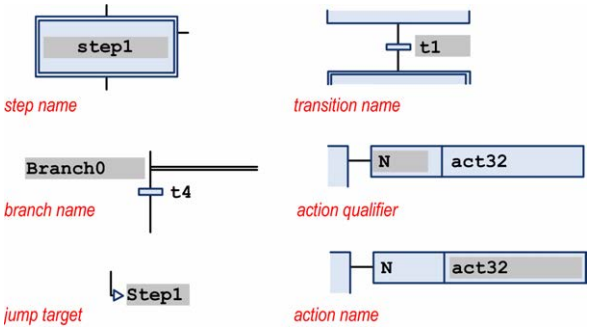
Overview

Possible cursor positions in an SFC diagram in the SFC editor, page 330 are indicated by a gray shadow when moving with the cursor over the elements.

Cursor Positions in Texts

There are 2 categories of cursor positions: texts and element bodies. See the possible positions indicated by a gray shaded area as shown in the following illustrations:

Possible cursor positions in texts:



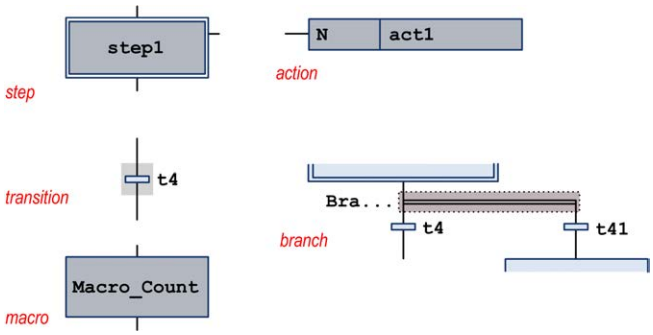
When you click a text cursor position, the string will become editable.

Select action name for editing:



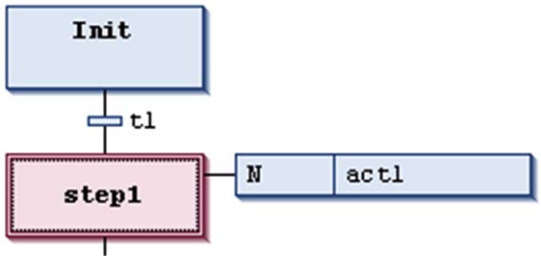
Cursor Positions in Element Bodies

Possible cursor positions in element bodies:



When you click a shadowed area, the element is selected. It gets a dotted frame and is displayed as red-shaded (for multiple selection, refer to *Working in the SFC Editor*, page 333).

Selected step element



Working in the SFC Editor

Overview

By default, a new SFC POU contains an initial step and a subsequent transition. This chapter provides information on how to add further elements, and how to arrange and edit the elements.

Possible Cursor Positions

For further information, refer to the chapter *Cursor Positions in SFC*, page 332.

Navigating

Use the arrow keys to jump to the next or previous element in the chart.

Inserting Elements

To insert the particular SFC elements, page 336, execute the respective commands from the **SFC** menu. For further information, refer to the description of the SFC editor commands (see Menu Commands, Online Help). Double-click an already inserted step, transition, or action element, which does not yet reference a programming object, to open a dialog box for assigning one.

Selecting Elements

Select an element or text field by clicking a possible cursor position. You can also give the selection to an adjacent element by using the arrow keys. The element will change color to red. For example, see the chapter *Cursor Positions in SFC*, page 332.

> **NOTE:** In contrast to previous versions of EcoStruxure Machine Expert, you can select and thus also move (cut, copy, paste) or delete steps and transitions separately.

For multiple selection, the following possibilities are available:

- Keep the SHIFT key pressed and then click the particular elements to be selected.
- Press the left mouse-key and draw a rectangle (dotted line) around the elements to be selected.
- Execute the command **Select All**, by default from the **Edit** menu.

Editing Texts

Click a text cursor position to open the edit field, where you can edit the text. If a text area has been selected via the arrow keys, open the edit field explicitly by using the SPACE bar.

Editing Associated Actions

Double-click a step (entry, active, or exit) or transition action association to open the associated action in the corresponding editor. You can, for example, double-click the transition element or the triangle indicating an exit action in a step element.

Cutting, Copying, Pasting Elements

Select the elements and execute the command **Cut**, **Copy**, or **Paste** (from the **Edit** menu) or use the corresponding keys.

NOTE:

- When you paste one or several cut or copied elements, the content of the clipboard will be inserted before the currently selected position. If nothing is selected, the elements will be appended at the end of the currently loaded chart.
- If you paste a branch while the currently selected element is also a branch, the pasted branch elements will be inserted to the left of the existing elements.
- If you paste an action (list) at a currently selected step, the actions will be added at the beginning of the action list of the step or an action list for the step will be created.
- Incompatible elements when cutting/copying:

  If you select an associated action (list) and additionally an element which is not the step to which the action (list) belongs, a message box will display: **The current selection contains incompatible elements. No data will be filed to the clipboard.** The selection will not be stored and you cannot paste or copy it somewhere else.
- Incompatible elements when pasting:

  If you try to paste an action (list) while the currently selected element is not a step or another association, a message box will display: **The current clipboard content cannot be pasted at the current selection.** If you try to paste an element like a step, branch, or transition when currently an associated action (list) is selected, the same message box will display.
- For copying step and transition elements that call action objects or transition objects, two different duplication modes are available. Either the references are copied only, or the referenced objects are embedded and duplicated when copying. Select a mode in the **Element Properties** dialog box with the **Duplicate on copy** option, page 336.

Deleting Elements

Select the elements and execute the command **Delete** or press the **Delete** key.

Consider the following:

- Deleting a step also deletes the associated action list.
- Deleting the initial step automatically sets the following step to be the initial one. This option **Initial step** will be activated in the properties of this step.
- Deleting the horizontal line preceding a branched area will delete all branches.
- Deleting all particular elements of a branch will delete the branch.

SFC Element Properties

Overview

You can view and edit the properties of an SFC element in the **Element Properties** dialog box. Open this dialog box via the command **Element Properties**, which is part of the **View** menu.

It depends on the currently selected element which properties are displayed.

You can configure whether the particular types of properties should be displayed next to an element in the SFC chart from the **View** tab of the SFC editor options.

Common Properties

Property	Description
Name	Element name, by default <element><running number> Examples: step name Step0, Step1, branch name branch0 and so on.
Comment	Element comment, text stringExample: Reset the counter. Press CTRL + ENTER to insert line breaks.
Symbol	For each SFC element implicitly a flag is created, named like the element. Here you can specify whether this flag variable should be exported to the symbol configuration and how the symbol then should be accessible in the controller. **None**: The symbol will be exported to the symbol configuration, but it will not be accessible in the controller. **Read**: The symbol will be exported to the symbol configuration and it will be readable in the controller. **Write**: The symbol will be exported to the symbol configuration and it will be written to the controller. **Read/Write**: Combination of read and write. By default, this field is left empty. That is that no symbol is exported to the symbol configuration.

Specific Properties

Specific Property		Description
Initial step		This option is activated in the properties of the current initial step (init step), page 336. By default, it is activated for the first step in an SFC and deactivated for other steps. If you activate this option for another step, you must deactivate it in the previous init step. Otherwise, a compiler error will be generated.
Duplicate on copy		This option is available for steps that contain a step action (entry action, main action, or exit action), and for transitions that are linked to a transition object. If the option **Duplicate on copy** is selected, a new object is created for each called action, step, and property when the step or transition is copied. The new object contains a copy of the implementation code of the source object. If the option **Duplicate on copy** is not selected, the link to the called object is retained for the associated actions, steps, and properties when the step or transition is copied. New objects are not created. The source and the copies of the step or transition call the same action, step, or property.
Times		Defines the minimum and maximum processing times for the step.
	Minimal active	Minimum length of time the processing of this step should take, even though the following transition is TRUE. Permissible values: time according to IEC syntax (for example, `t#8s`) or TIME variable; default: `t#0s`.
	Maximal active	Maximum length of time the processing of this step should take. Time outs in steps are indicated by the implicit variable, page 347 *SFCError* flag. Permissible values: time according to IEC syntax (for example, `t#8s`) or TIME variable; default: `t#0s`
Actions		Defines the actions, page 340 to be performed when the step is active. Refer to the description of the Sequence of Processing in SFC, page 353 for details.
	Step entry	This action will be executed after the step has become active.
	Step active	This action will be executed when the step is active and possible entry actions have already been processed.
	Step exit	This action will be executed in the subsequent cycle after a step has been deactivated (exit action).

NOTE: Use the corresponding implicit variables to determine the status of actions and time outs via SFC flags, page 347.

SFC Elements / ToolBox

Overview

You can insert the graphic elements usable for programming in the SFC editor window by executing the commands from the **SFC** menu.

For information on working in the editor, refer to the description in the chapter *Working in the SFC Editor*, page 333

The following elements are available and are described in this chapter:

- step, page 337
- transition, page 337

- action, page 340
- branch (alternative), page 343
- branch (parallel), page 344
- jump, page 345
- macro, page 346

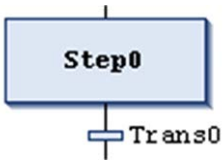
Step / Transition

To insert a single step or a single transition, execute the command **Step** or **Transition** from the **ToolBox**. Steps and transitions can also be inserted in combination, via command **Insert step-transition** ( ) or **Insert step-transition after** ( ) from the toolbar.

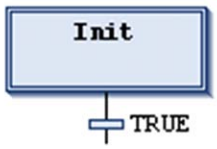
A step is represented by a box primarily containing an automatically generated step name. It is connected to the preceding and subsequent transition by a line. The box frame of the first step within an SFC, the initial step, is double-lined.

The transition is represented by a small rectangle. After inserting it has a default name, `Trans<n>`, whereby `n` is a running number.

Example for step and subsequent transition:



Example for initial step and subsequent transition:



You can edit the step and transition names inline.

Step names must be unique in the scope of the parent POU. Consider this especially when using actions programmed in SFC. Otherwise an error will be detected during the build process.

You can transform each step to an initial step by executing the command **Init step** or by activating the respective step property. An initial step will be executed first when the IL-POU is called.

Each step is defined by the step properties, page 335.

After you have inserted a step, associate the actions to be performed when the step is active (processed); see below for further information on actions, page 340.

About Transitions

A transition has to provide the condition on which the subsequent step shall become active as soon as the condition value is TRUE. Therefore, a transition condition must have the value TRUE or FALSE.

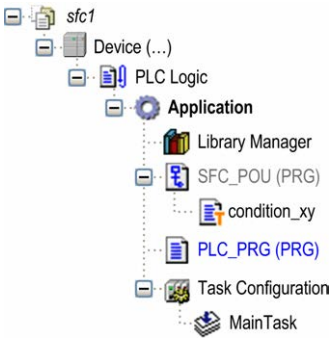
A transition condition can be defined in the following 2 ways:

Type of Definition	Type of Condition	Description
direct	inline	Replace the default transition name by one of the following elements: • boolean variable • boolean address • boolean constant • instruction having a boolean result (example: (i<100) AND b). You cannot specify programs, function blocks, or assignments here.
using a separate transition or property object	multi-use	Replace the default transition name by the name of a transition ( ) or property object ( ) available in the project. (This allows multiple use of transitions; see for example `condition_xy` in the figures below.) The object like an inline transition can contain the following elements: • boolean variable • address • constant • instruction • multiple statements with arbitrary code

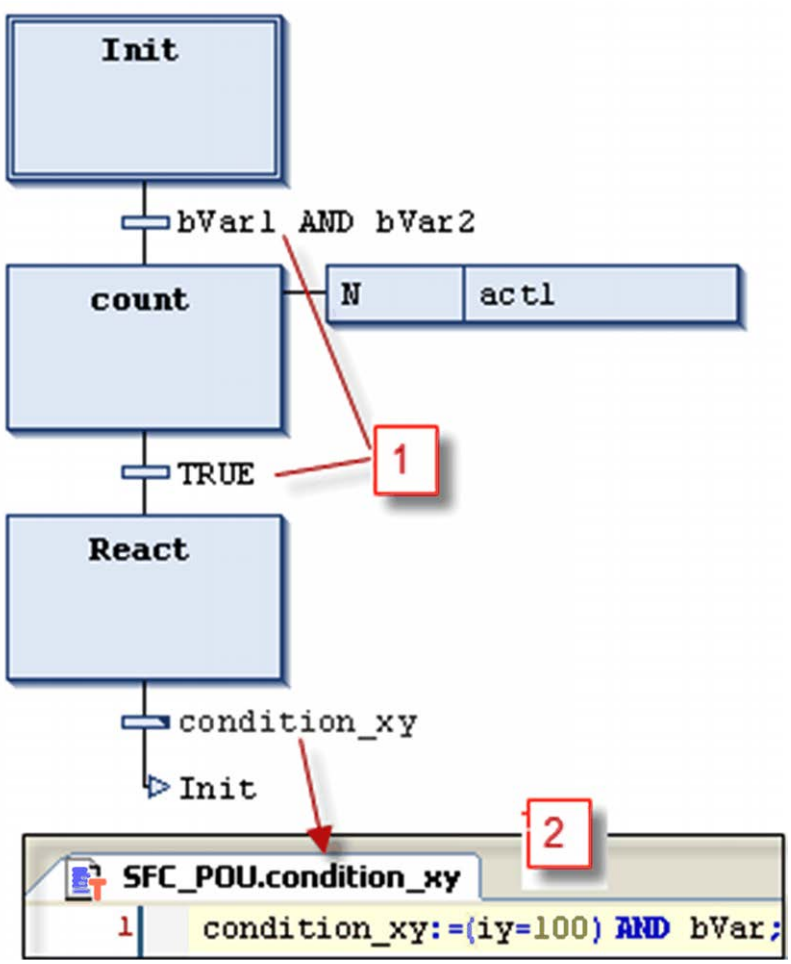
NOTE: If a transition produces multiple statements, assign the desired expression to a transition variable.

NOTE: Transitions which consist of a transition or a property object are indicated by a small triangle in the upper right corner of the rectangle.

Transition object (multiple use transition):



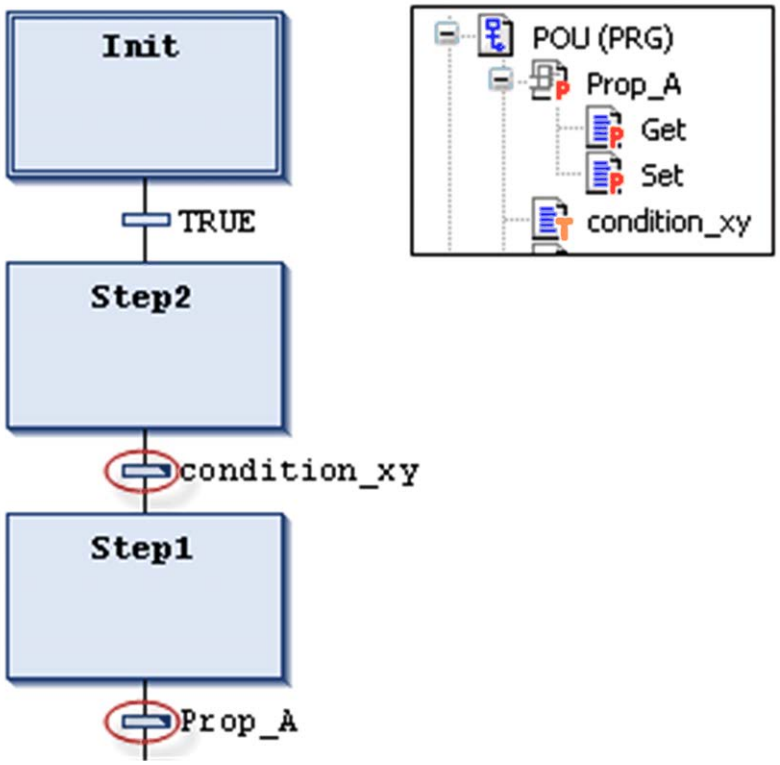
Examples of transitions:



1 Transition conditions entered directly

2 Transition `condition_xy` programmed in ST

Multiple use conditions (transitions or properties) are indicated by a triangle:



In contrast to previous versions of EcoStruxure Machine Expert, a transition call is handled like a method call. It will be entered according to the following syntax:

<transition name>:=<transition condition>;

Example: `trans1:= (a=100);`

or just

<transition condition>;

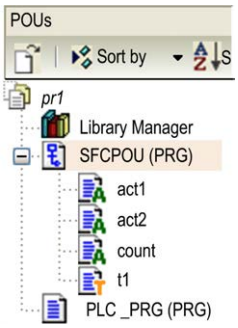
Example: `a=100;`

See also an example (`condition_xy`) in the figure *Examples of transitions*.

Action

An action can contain one or more statements written in one of the valid programming languages. It is assigned to a step and, in online mode, it will be processed according to the defined sequence of processing, page 353 .

Each action to be used in SFC steps must be available as a valid POU within the SFC POU or the project ( ).

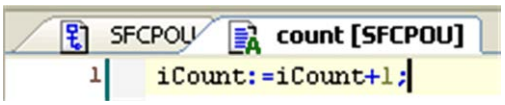


If you add IEC actions to a step as action association (see Menu Commands, Online Help), you can also specify a Boolean variable instead of an action object. The value of these variables is toggled between FALSE and TRUE each time the action is executed.

NOTE: When associating a boolean variable to an IEC step action, do not use this boolean variable at another place throughout this SFC POU.

Step names must be unique in the scope of the parent POU. An action may not contain a step having the same name as the step to which it is assigned; otherwise, an error will be detected during the build process.

Example of an action written in ST



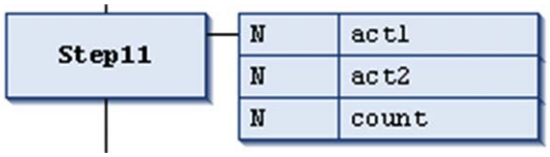
The IEC conforming and the IEC extending step actions are described in the following paragraphs.

IEC Conforming Step Action (IEC Action)

This is an action according to the IEC61131-3 standard which will be processed according to its qualifier, page 346 when the step becomes active, and then a second time when it becomes deactivated. In case of assigning multiple actions to a step, the action list will be executed from top to bottom.

- Different qualifiers can be used for IEC step actions in contrast to a normal step action.
- A further difference to the normal step actions is that each IEC step action is provided with a control flag. This permits that, even if the action is called also by another step, the action is executed always only once at a time. This is not the case with the normal step actions.
- An IEC step action is represented by a bipartite box connected to the right of a step via a connection line. In the left part, it shows the action qualifier, in the right part the action name. You can both edit inline.
- IEC step actions are associated to a step via the **Insert action association** command. You can associate one or multiple actions with a step. The position of the new action depends on the current cursor position and the command. The actions have to be available in the project and be inserted with a unique action name (for example, `plc_prg.a1`).

IEC conforming step action list associated to a step:



Each action box in the first column shows the qualifier and in the second the action name.

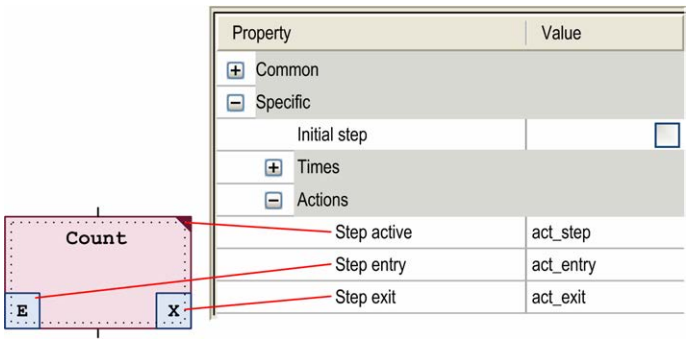
IEC Extending Step Actions

These are actions extending the IEC standard. They have to be available as objects below the SFC object. Select unique action names. They are defined in the step properties.

The table lists the IEC extending step actions:

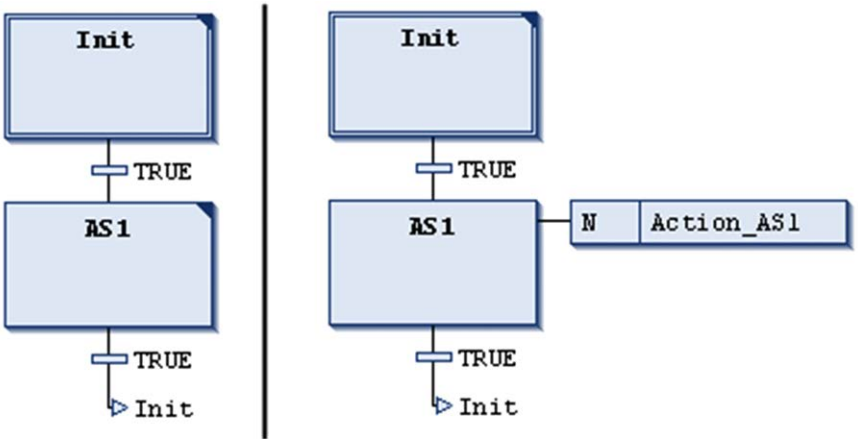
Action Type	Processing	Association	Representation
step entry action (step activated)	This type of step action will be processed as soon as the step has become active and before the step active action.	The action is associated to a step via an entry in the **Step entry** field of the step properties, page 335.	It is represented by an `E` in the lower left corner of the respective step box.
step active action (step action)	This type of step action will be processed when the step has become active and after a possible step entry action of this step has been processed. However, in contrast to an IEC step action (see above) it is not executed again when it is deactivated and cannot get assigned qualifiers.	The action is associated to a step via an entry in the **Step active** field of the step properties, page 335.	It is represented by a small triangle in the upper right corner of the respective step box.
step exit action (step deactivated)	An exit action will be executed once when the step becomes deactivated. However, this execution will not be done in the same, but at the beginning of the subsequent cycle.	The action is associated to a step via an entry in the **Step exit** field of the step properties, page 335.	It is represented by an `X` in the lower right corner of the respective step box.

IEC extending step actions



Example: Differences Between IEC Matching / Extending Step Actions

The main difference between step actions and IEC actions with qualifier `N` is that the IEC action is at least executed twice: first time when the step is active and the second time when the step is deactivated. See the following example.



Action `Action_AS1` is associated to step `AS1` as a step action (left), or as an IEC action with qualifier `N` (right). Due to the fact that in both cases 2 transitions are used, it will take 2 controller cycles each before the initial step is reached again, assuming that a variable `iCounter` is incremented in `Action_AS1`. After a reactivation of step `Init`, `iCounter` in the left example will have value 1. In the right one however, it will have value 2 because the IEC action - due to the deactivation of `AS1` - has been executed twice.

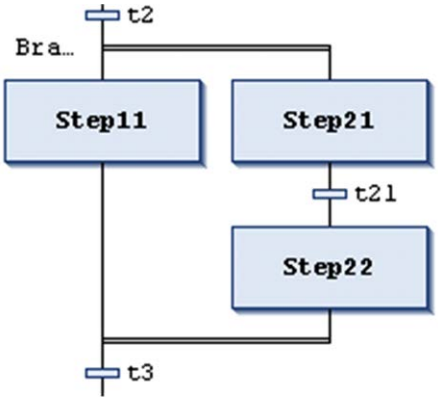
For further information on qualifiers, refer to the list of available qualifiers, page 347.

In contrast to IEC actions, step actions can be embedded in such a way that they can only be called from the corresponding step. If you copy this step, new action objects are automatically created and the implementation code is duplicated. Select the duplication mode for step actions as **Copy reference** or **Copy implementation** in the dialog box that is displayed when you insert the first action into the step. You can select the duplication mode later by activating the **Duplicate on copy** option, page 336 in the **Element Properties** dialog box. It is also possible to configure the **Default insertion method** as a general option in the **Tools > Options > SFC Editor** dialog box (see Menu Commands, Online Help).

Branches

A sequential function chart (SFC) can diverge; that is the processing line can be branched into 2 or several further lines (branches). Parallel branches, page 344 will be processed parallel (simultaneously). In the case of alternative branches, page 343, only one will be processed depending on the preceding transition condition. Each branching within a chart is preceded by a horizontal double (parallel) or simple (alternative) line and also terminated by such a line or by a jump, page 345.

Parallel Branch



A parallel branch has to begin and end with a step. Parallel branches can contain alternative branches or other parallel branches.

The horizontal lines before and after the branched area are double-lines.

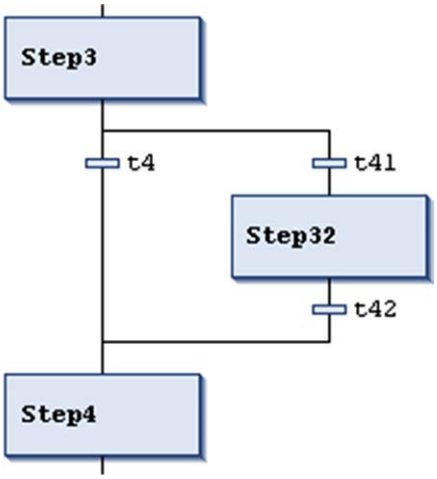
Processing in online mode: If the preceding transition (`t2` in the example shown on the left) is TRUE, the first steps of all parallel branches will become active (`Step11` and `Step21`). The particular branches will be processed in parallel to one another before the subsequent transition (`t3`) will be recognized.

To insert a parallel branch, select a step and execute the command **Insert branch right**.

You can transform parallel and alternative branches to each other by executing the commands **Parallel** or **Alternative**.

Automatically a branch label is added at the horizontal line preceding the branching which is named `Branch<n>` whereby `n` is a running number starting with 0. You can specify this label when defining a jump target, page 345.

Alternative Branch



The horizontal lines before and after the branched area are simple lines.

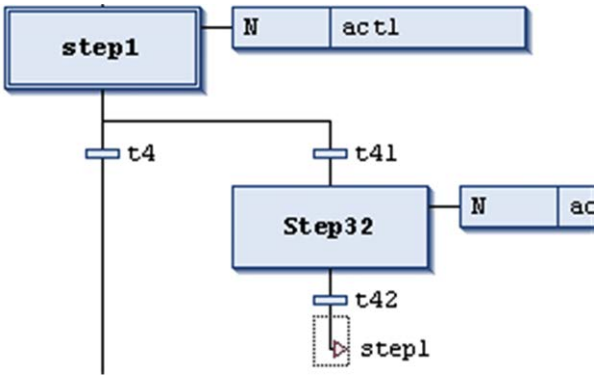
An alternative branch has to begin and end with a transition. Alternative branches can contain parallel branches and other alternative branches.

If the step which precedes the alternative beginning line is active, then the first transition of each alternative branch will be evaluated from left to right. The first transition from the left whose transition condition has value TRUE, will be opened, and the following steps will be activated.

To insert alternative branches, select a transition and execute the command **Insert branch right**.

You can transform parallel and alternative branches from one another by executing the commands **Parallel** or **Alternative**.

Jump



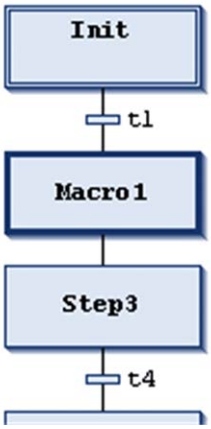
A jump is represented by a vertical connection line plus a horizontal arrow and the name of the jump target. It defines the next step to be processed as soon as the preceding transition is TRUE. You can use jumps to avoid that processing lines cross or lead upward.

Besides the default jump at the end of the chart, a jump may only be used at the end of a branch. To insert a jump, select the last transition of the branch and execute the command **Insert jump**.

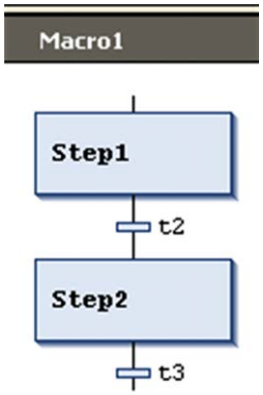
The target of the jump is specified by the associated text string which can be edited. It can be a step name or the label of a parallel branch.

Macro

Main SFC editor view



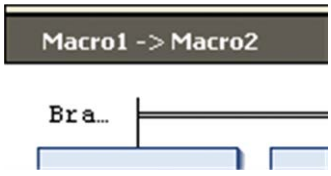
Macro editor view for `Macro1`



A macro is represented by a bold-framed box containing the macro name. It includes a part of the SFC chart, which thus is not directly visible in the main editor view. The process flow is not influenced by using macros; it is just a way to hide some parts of the program, for example, in order to simplify the display. To insert a macro box, execute the command **Insert macro (after)**. The macro name can be edited.

To open the macro editor, double-click the macro box or execute the command **Zoom into macro**. You can edit here just as in the main editor view and enter the desired section of the SFC chart. To get out, execute the command **Zoom out of macro**.

The title line of the macro editor shows the path of the macro within the current SFC example:



Qualifier for Actions in SFC

Overview

In order to configure in which way the actions, page 340 should be associated to the IEC steps, some qualifiers are available, which are to be inserted in the qualifier field of an action element.

Available Qualifiers

Qualifier	Long Form	Description
N	non-stored	The action is active as long as the step is active.
R	overriding reset	The action becomes deactivated.
S	set (stored)	The action will be started when the step becomes active and will be continued after the step is deactivated until the action is reset.
L	time limited	The action will be started when the step becomes active. It will continue until the step becomes inactive or a set time has passed.
D	time delayed	A delay timer will be started when the step becomes active. If the step is still active after the time delay, the action will start and continue until it is deactivated. **NOTE:** When two consecutive steps have a D (time delayed) action for setting the same boolean variable, this variable will not be reset during the transition from one step to the other. In order to reset the variable, insert an intermediate step between the two steps.
P	pulse	The action will be started when the step becomes active/deactive.
SD	stored and time delayed	The action will be started after the set time delay and it will continue until it is reset.
DS	delayed and stored	If the step is still active after the specified time delay, the action will start and it will continue until it is reset.
SL	stored and time limited	The action will be started when the step becomes active and it will continue for the specified time or until a reset.

The qualifiers L, D, SD, DS, and SL need a time value in the TIME constant format. Enter it directly after the qualifier, separated by a blank space, for example `L T#10s`.

NOTE: When an IEC action has been deactivated, it will be executed one more time. The implication is that each action will execute at least twice. This also applies to actions with P qualifier.

Implicit Variables - SFC Flags

Overview

Each SFC step and IEC action provides implicitly generated variables for watching the status, page 348 of steps and IEC actions during runtime. Also, you can define variables for watching and controlling the execution of an SFC (timeouts, reset, tip mode). These variables can also be generated implicitly by the SFC object.

Basically, for each step and each IEC action, an implicit variable is generated. A structure instance, named for the element, for example, step1 for a step with step name **step1**. You can define in the element properties, page 335 whether for this flag a symbol definition should be exported to the symbol configuration and how this symbol should be accessible in the controller.

The data types for those implicit variables are defined in library IecSFC.library. This library will automatically be included in the project as soon as an SFC object is added.

Step and Action Status and Step Time

Basically, for each step and each IEC action, an implicit structure variable of type *SFCStepType* or *SFCActionType* is created. The structure components (flags) describe the status of a step or action or the currently processed time of an active step.

The syntax for the implicitly done variable declaration is:

<stepname>: SFCStepType;

or

_<actionname>:SFCActionType;

NOTE: Implicit variables for IEC actions are preceded by an underscore.

The following boolean flags for step or action states are available:

Boolean flags for **steps**:

Boolean Flag	Description
<stepname>.x	shows the current activation status
<stepname>._x	shows the activation status for the next cycle

If <stepname>.x = TRUE, the step will be executed in the current cycle.

If <stepname>._x = TRUE and <stepname>.x = FALSE, the step will be executed in the following cycle. Therefore, <stepname>._x is copied to <stepname>.x at the beginning of a cycle.

Boolean flags for **actions**:

Boolean Flag	Description
_<actionname>.x	is TRUE if the action is executed
_<actionname>._x	is TRUE if the action is active

Symbol generation

In the element properties, page 335 of a step or an action, you can define if a symbol definition should be added to a possibly created and downloaded symbol configuration for the step or action name flag. For this purpose, make an entry for the desired access right in column **Symbol** of the element properties view.

NOTE: If you use the boolean flag <stepname>.x to force a certain status value for a step (for setting a step active), be aware that this will affect uncontrolled states within the SFC.

⚠WARNING

UNINTENDED EQUIPMENT OPERATION

Do not use the boolean flag <stepname>.x to force a status value for setting a step active.

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

Time via TIME variables:

The flag `t` gives the current time span which has passed since the step had become active. This is only for steps, no matter whether there is a minimum time configured in the step attributes, page 335 or not (see below: *SFCError*).

For **steps**:

<stepname>.t (<stepname>._t not usable for external purposes)

For **actions**:

The implicit time variables are not used.

Control of SFC Execution (Timeouts, Reset, Tip Mode)

You can use some implicitly available variables, also named SFC flags (see table below) to control the operation of an SFC. For example, for indicating time overflows or enabling tip mode for switching transitions.

In order to be able to access these flags you have to declare and activate them. Do this in the **SFC Settings** dialog box. This is a subdialog box of the object **Properties** dialog box.

Manual declaration, as it was needed in SoMachine / SoMachine Motion V3.1, is only necessary to enable write access from another POU (refer to the paragraph *Accessing Flags*).

In this case, consider the following:

If you declare the flag globally, you have to deactivate the **Declare** option in the **SFC Settings** dialog box.Otherwise this leads to an implicitly declared local flag, which then would be used instead of the global one. Keep in mind, that the SFC settings for an SFC POU initially are determined by the definitions set in the **Options > SFC** dialog box.

Consider that a declaration of a flag variable solely done via the **SFC Settings** dialog box will only be visible in the online view of the SFC POU.

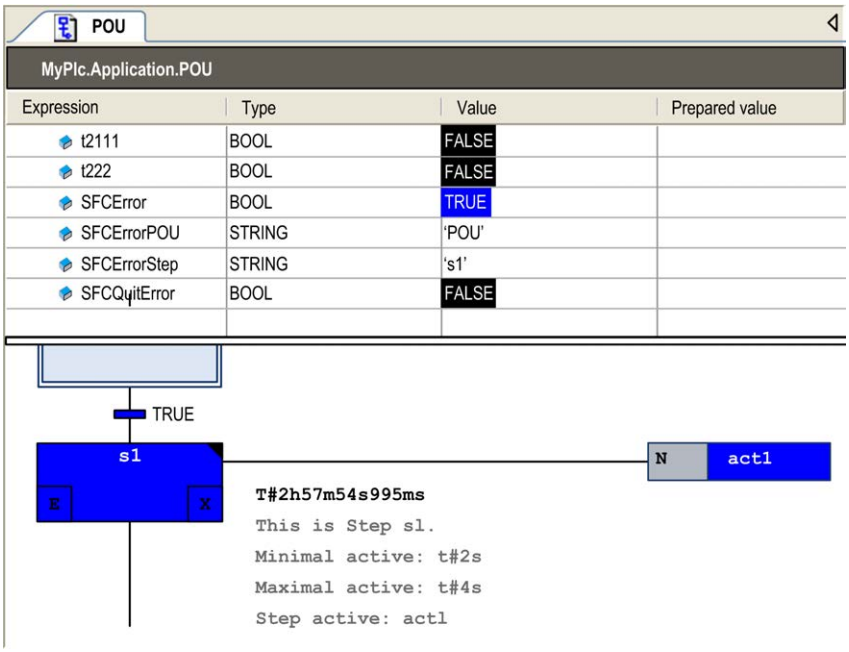
The following implicit variables (flags) can be used. For this purpose, you have to declare and activate them in the **SFC Settings** dialog box.

Variable	Type	Description
SFCInit	BOOL	If this variable becomes TRUE, the sequential function chart will be set back to the Init step, page 336. All steps and actions and other SFC flags will be reset (initialization). The initial step will remain active, but not be executed as long as the variable is TRUE. Set back *SFCInit* to FALSE in order to get back to normal processing.
SFCReset	BOOL	This variable behaves similarly to *SFCInit*. Unlike the latter however, further processing takes place after the initialization of the initial step. Thus, in this case, a reset to FALSE of the *SFCReset* flag could be done in the initial step.
SFCError	BOOL	As soon as any timeout occurs at 1 of the steps in the SFC, this variable will become TRUE. Precondition: *SFCEnableLimit* must be TRUE. Consider that any further timeout cannot be registered before a reset of *SFCError*. *SFCError* must be defined, if you want to use the other time-controlling flags (*SFCErrorStep*, *SFCErrorPOU*, *SFCQuitError*). **NOTE:** You can use the flags *SFCErrorAnalyzation* and *SFCErrorAnalyzationTable* to determine the components of the expression that contributes to the value TRUE of the *SFCError*. In the SFC editor, the flag *SFCErrorAnalyzationTable* uses this function implicitly for the analysis of expressions in transitions.
SFCEnableLimit	BOOL	You can use this variable for the explicit activation (TRUE) and deactivation (FALSE) of the time control in steps via *SFCError*. Therefore, if this variable is declared and activated (**SFC Settings**) then it must be set TRUE in order to get *SFCError* working. Otherwise, any timeouts of the steps will not be registered. The usage can be reasonable during start-ups or at manual operation. If the variable is not defined, *SFCError* will work automatically. Precondition: *SFCError* must be defined.
SFCErrorStep	STRING	This variable stores the name of a step at which a timeout was registered by *SFCError.timeout*. The name is stored until the registered step error is reset by means of *SFCQuitError*. Precondition: *SFCError* must be defined.
SFCErrorPOU	STRING	This variable stores the name of the SFC POU in which a timeout has occurred. The name is stored until the timeout is reset by means of *SFCQuitError*. Precondition: *SFCError* must be defined.
SFCQuitError	BOOL	As long as this variable is TRUE, the execution of the SFC diagram is stopped, and variable *SFCError* will be reset. As soon as the variable has been reset to FALSE, all current time states in the active steps will be reset. Precondition: *SFCError* must be defined.
SFCPause	BOOL	As long as this variable is TRUE, the execution of the SFC diagram is stopped.
SFCTrans	BOOL	This variable becomes TRUE, as soon as a transition is actuated.
SFCCurrentStep	STRING	This variable stores the name of the currently active step, independently of the time monitoring. In case of simultaneous sequences, the name of the outer right step will be registered.
SFCTipSFCTip-Mode	BOOL	These variables allow using the inching mode within the current chart. When this mode has been switched on by *SFCTipMode*=TRUE, you can only skip to the next step by setting *SFCTip*=TRUE (rising edge). As long as *SFCTipMode* is set to FALSE, it is possible to skip by the transitions.

Variable	Type	Description
SFCErrorAnaly-zation	–	String variable that contains the variables contributing to the total value TRUE of *SFCError* (timeout in one step). As a prerequisite, *SFCError* must be activated. *SFCErrorAnalyzation* implicitly uses the function of the POU *AnalyzeExpression* of the Analyzation library.
SFCErrorAnaly-zationTable	–	Table that contains the variables contributing to the total value TRUE of *SFCError* (timeout in one step). As a prerequisite, *SFCError* must be activated. *SFCErrorAnalyzationTable* implicitly uses the function of the POU *AnalyzeExpressionTable* of the Analyzation library.

The following figure provides an example of some SFC detected error flags in online mode of the editor.

A timeout has been detected in step *s1* in SFC object **POU** by flag *SFCError*.



Accessing Flags

For enabling access on the flags for the control of SFC execution (timeouts, reset, tip mode), declare and activate the flag variables as described above (Control of SFC Execution, page 349).

Syntax for accessing from an action or transition within the SFC POU:

<stepname>.<flag>

or

_<actionname>.<flag>

Examples:
```
status:=step1._x;
checkerror:=SFCerror;
```

Syntax for accessing from another POU:

<SFC POU>.<stepname>.<flag>

or

<SFC POU>._<actionname>.<flag>

Examples:
```
status:=SFC_prog.step1._x;
checkerror:=SFC_prog.SFCerror;
```

Consider the following in case of write access from another POU:

- The implicit variable additionally has to be declared explicitly as a VAR_INPUT variable of the SFC POU
- or it has to be declared globally in a GVL (global variable list).

Example: Local declaration
```
PROGRAM SFC_prog
VAR_INPUT
  SFCinit:BOOL;
END_VAR
```

Example: Global declaration in a GVL
```
VAR_GLOBAL
  SFCinit:BOOL;
END_VAR
```

Accessing the flag in `PLC_PRG`:
```
PROGRAM PLC_PRG
VAR
  setinit: BOOL;
END_VAR
SFC_prog.SFCinit:=setinit;  //Write access to SFCinit in
SFC_prog
```

Analyzation of Expressions

The Analyzation library allows to analyze expressions. It can be used, for example, in the SFC diagram to examine the result of the flag *SFCError*. This flag is used to monitor timeouts in the SFC diagram.

You can use the flags *SFCErrorAnalyzation* and *SFCErrorAnalyzationTable* to determine the components of the expression that contributes to the value TRUE of the *SFCError*.

Sequence of Processing in SFC

Overview

In online mode, the particular action types will be processed according a defined sequence; see the table below.

Definition of Terms

The following terms are used:

Term	Description
active step	A step, whose step action is being executed. In online mode active steps are filled with blue color.
initial step	In the first cycle after an SFC POU has been called, the initial step automatically becomes active and the associated step action, page 340 is executed.
IEC actions	IEC actions are executed at least twice: • The first time when they became active. • The second time - in the following cycle - when they have been deactivated.
alternative branches	If the step preceding the horizontal start line of alternative branches is active, then the first transition of each particular branch will be evaluated from left to right. The first transition from the left whose transition condition has value TRUE will be searched and the respective branch will be executed that is the subsequent step within this branch will become active.
parallel branches	If the double-line at the beginning line of parallel branches is active and the preceding transition condition has the value TRUE, then in all parallel branches each first step will become active. The branches now will be processed in parallel to one another. The step subsequent to the double-line at the end of the branching will become active when all previous steps are active and the transition condition after the double-line has the value TRUE.

Processing Order

Processing order of elements in a sequence:

Step	Description
1. Reset of the IEC Actions	All action control flags of the IEC actions, page 340 get reset (not, however, the flags of IEC actions that are called within actions).
2. Step exit actions (step deactivated)	All steps are checked in the order which they assume in the sequence chart (top to bottom and left to right) to determine whether the requirement for execution of the step exit action is provided.If that is the case, it will be executed. An exit action will be executed if the step is going to become deactivated, page 337. This means if the entry and step actions - if existing - have been executed during the last cycle, and if the transition for the following step is TRUE.
3. Step entry actions(step activated)	All steps are tested in the order which they assume in the sequence to determine whether the requirement for execution of the step entry action is provided. If that is the case, it will be executed. An entry action will be executed if the step-preceding transition condition is TRUE and thus the step has been activated.
4. Timeout check, step active actions	For non-IEC steps, the corresponding step active action is now executed in the order in which they are positioned in the sequence (top -> down and left -> right).

Step	Description
5. IEC actions	IEC actions, page 340 that are used in the sequence are executed in alphabetical order. This is done in 2 passes through the list of actions. In the first pass, all the IEC actions that are deactivated in the current cycle are executed. In the second pass, all the IEC actions that are active in the current cycle are executed.
6. Transition check, activating next steps	Transitions, page 337 are evaluated. If the step in the current cycle was active and the following transition returns TRUE (and if applicable the minimum active time has already elapsed), then the following step is activated.

NOTE: An action may be executed multiple times in 1 cycle because it is called from more than one other IEC actions when there are multiple steps active. That is to say, the same IEC action is used simultaneously in different levels of an SFC, and this could lead to undesired effects.

Example: An SFC could have 2 IEC actions A and B, which are both implemented in SFC, and which both call IEC action C. Then in IEC actions A and B both can be active in the same cycle and furthermore, in both actions IEC action C can be active. In this case, C would be called twice.

⚠WARNING

UNINTENDED EQUIPMENT OPERATION

Do not call IEC actions from multiple other IEC actions in the same cycle.

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

NOTE: Use implicit variables, page 347 for determining the status of steps and actions or the execution of the chart.

SFC Editor in Online Mode

Overview

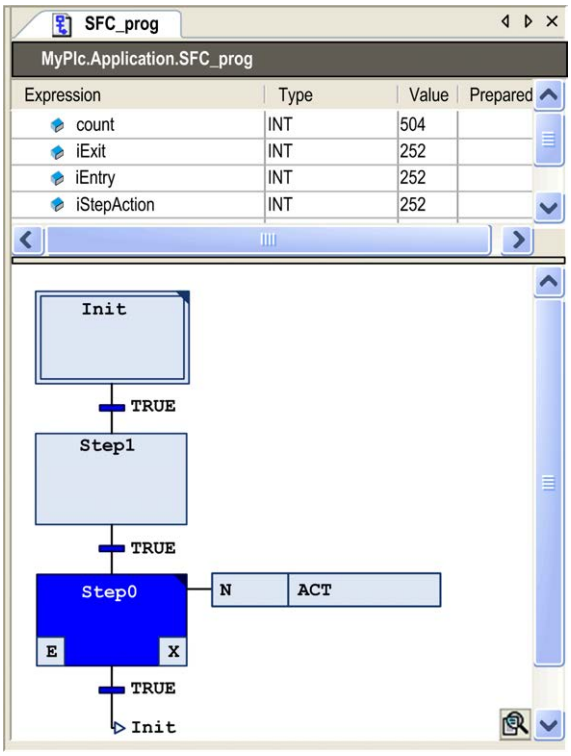
In online mode, the SFC editor provides views for monitoring (see below) and for writing and forcing the variables and expressions on the controller. Debugging functionality like for the other IEC languages (breakpoints, stepping, and so on) is not available in SFC. However, consider the following hints for debugging SFC:

- For information on how to open objects in online mode, refer to the description of the user interface in online mode, page 50.
- The editor window of an SFC object also includes the declaration editor in the upper part. For general information, refer to the chapter *Declaration Editor in Online Mode*, page 378 . If you have declared implicit variables (SFC flags), page 347 via the SFC **Settings** dialog box, they will be added here, but will not be viewed in the offline mode of the declaration editor.
- Consider the sequence of processing, page 353 of the elements of a Sequential Function Chart.
- See the object properties or the SFC editor options and SFC defaults for settings concerning compilation or online display of the SFC elements and their attributes.
- Consider the possible use of flags, page 347 for watching and controlling the processing of an SFC.

Monitoring

Active steps are displayed as filled with blue color. The display of step attributes depends on the set SFC editor options.

Online view of program object `SFC_prog`



Structured Text (ST) Editor

What's in This Chapter

Information on the ST Editor 356
Structured Text ST 360

Information on the ST Editor

ST Editor

Overview

The ST editor is used to create programming objects in the IEC programming language Structured Text (ST) or extended Structured Text which provides some extensions to the IEC 61131-3 standard.

The ST editor is a text editor. Therefore, use the corresponding text editor settings in the **Options** and **Customize** dialog boxes to configure behavior, appearance, and menus. There you can define the default settings for highlight coloring, line numbers, tabs, indenting, and many other options.

The following functions are provided to help programming:

- List components functionality (to be activated using the **Tools > Options > Smart Coding** dialog box (see Menu Commands, Online Help)
- **Input Assistant**, page 429
- Code formatting functionality that is started by running the **Code Formatting > Format Code** (or **Format Code (selection)**) command from the **Edit** menu or from the contextual menu that opens upon right-clicking in the declaration or implementation part of the ST editor view. To configure this functionality, use the **Tools > Options > Code Formatter** dialog box (see Menu Commands, Online Help).

Further Information

- To scroll within the editor view, hold down the **Shift** key and use the scroll wheel of the mouse.
- To zoom within the editor view, hold down the **Ctrl** key and use the scroll wheel of the mouse.
- To select blocks, hold down the **ALT** key and select the desired text area with the mouse.

The editor will be available in the lower part of a window which also includes the declaration editor, page 373 in the upper part.

If syntactic errors are detected during editing, the corresponding messages will be displayed in the **Precompile Messages** window. This window is updated each time you reset the focus to the editor window (for example, place your cursor in another window and then back to the editor window).

Searching for Strings

To perform an incremental search for strings in the editor, proceed as follows:

Step	Action	Results
1	Press the keys **Ctrl** + **Shift** + **i**.	An input field at the bottom edge of the ST editor opens.
2	Type characters.	• The strings matching the characters are highlighted in the editor. • The number of matches detected is displayed on the right-hand side of the input field.
3	Use the arrow keys or the keyboard shortcuts **Alt** + **Page up** or **Alt** + **Page down** to navigate through the matching locations in the editor.	–

ST Editor in Online Mode

Overview

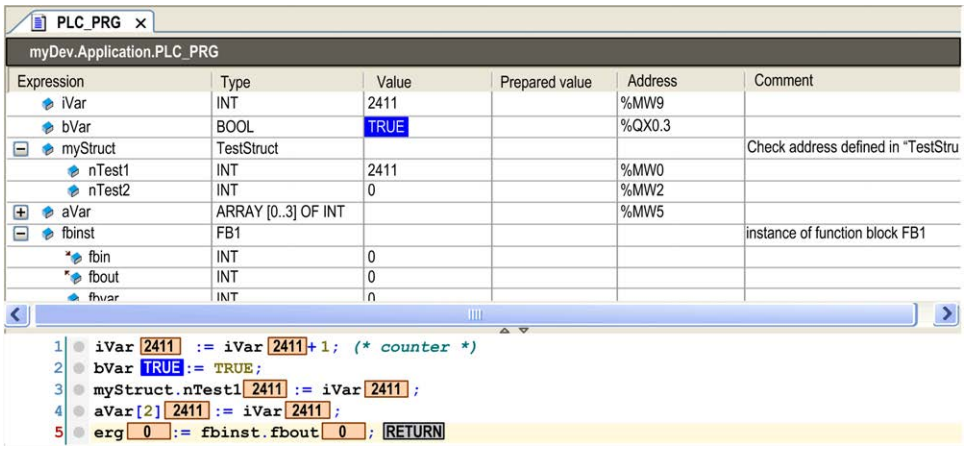
In online mode, the structured text editor (ST editor) provides views for monitoring, page 357, and for writing, and forcing the variables and expressions on the controller. Debugging (breakpoints, stepping, and so on) is available. See Breakpoint positions in ST Editor, page 359.

- For information on how to open objects in online mode, refer to the description of the user interface in online mode, page 50.
- For information on how to enter prepared values for variables in online mode, see *Forcing of variables*, page 358.
- The editor window of an ST object also includes the declaration editor in the upper part. For information on the declaration editor in online mode, see Declaration Editor in Online Mode, page 378.

Monitoring

If the **Enable inline monitoring** option is activated in the **Monitoring** tab of the **Tools > Options > Text editor** dialog box, small monitoring boxes will be displayed behind each variable showing the actual value.

Online view of a program object `PLC_PRG` with monitoring:



Forcing of Variables

In addition to the possibility of entering a prepared value for a variable within the declaration of any editor, the ST editor offers double-clicking the monitoring box of a variable within the implementation part (in online mode). Enter the prepared value in the dialog box that appears.

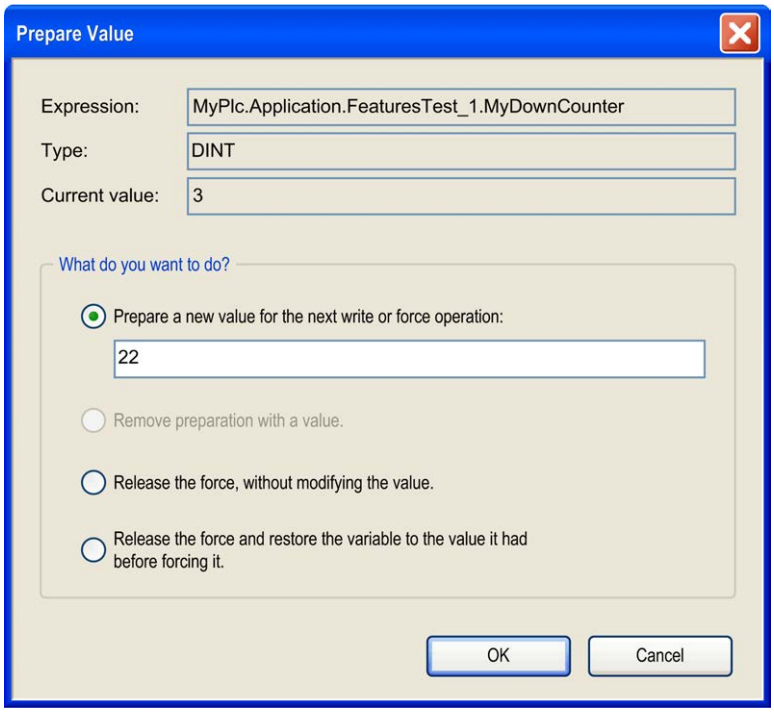
⚠WARNING

UNINTENDED EQUIPMENT OPERATION

- You must have a thorough understanding of how forcing will affect the outputs relative to the tasks being executed.
- Do not attempt to force I/O that is contained in tasks that you are not certain will be executed in a timely manner, unless your intent is for the forcing to take affect at the next execution of the task whenever that may be.
- If you force an output and there is no apparent affect on the physical output, do not exit the online mode without removing the forcing.
- If the online mode was interrupted while forcing was active, re-establish the connection with the controller and remove the forcing.

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

Dialog box **Prepare Value**



You find the name of the variable completed by its path within the **Devices Tree** (**Expression**), its type, and current value.

By activating the corresponding item, you can choose the following options:

- preparing a new value which has to be entered in the edit field
- removing a prepared value
- releasing the variable that is being forced
- releasing the variable that is being forced and resetting it to the value it was assigned before forcing

To carry out the selected action, execute the command **Debug > Force values** (item **Online**) or press the F7 key.

Breakpoint Positions in ST Editor

You can set a breakpoint basically at the positions in a POU where values of variables can change or where the program flow branches out or another POU is called. In the following descriptions, `{BP}` indicates a possible breakpoint position.

Assignment:

At the beginning of the line. Keep in mind that assignments as expressions, page 360 define no further breakpoint positions within a line.

FOR-loop:

1. before the initialization of the counter
2. before the test of the counter
3. before a statement

```
{BP} FOR i := 12 TO {BP} x {BP} BY 1 DO
{BP} [statement1]
...
{BP} [statementn-2]
END_FOR
```

WHILE-loop:

1. before checking the condition
2. before an instruction

```
{BP} WHILE i < 12 DO
{BP} [statement1]
...
{BP} [statementn-1]
END_WHILE
```

REPEAT-loop:

- before checking the condition

```
REPEAT
{BP} [statement1]
...
{BP} [statementn-1]
{BP} UNTIL i >= 12
END_REPEAT
```

Call of a program or a function block:

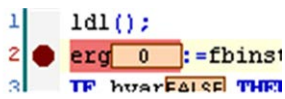
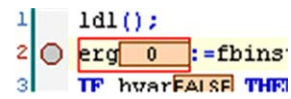
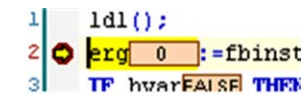
At the beginning of the line.

```
{{BP} POU( );
```

At the end of a POU:

When stepping through, this position will also be reached after a *RETURN* instruction.

Breakpoint display in ST

Breakpoint in Online Mode	Disabled Breakpoint	Program Stop at Breakpoint
		

NOTE: A breakpoint will be set automatically in all methods which may be called. If an interface-managed method is called, breakpoints will be set in all methods of function blocks implementing that interface and in all derivative function blocks subscribing the method. If a method is called via a pointer on a function block, breakpoints will be set in the method of the function block and in all derivative function blocks which are subscribing to the method.

Structured Text ST

Structured Text ST

Overview

Structured Text is a textual high-level programming language, similar to PASCAL or C. The program code is composed of expressions, page 360 and instructions, page 362. In contrast to IL (Instruction List), you can use numerous constructions for programming loops, thus allowing the development of complex algorithms.

Example

```
IF value < 7 THEN
 WHILE value < 8 DO
  value:=value+1;
 END_WHILE;
END_IF;
```

Expressions

Overview

An expression is a construction which after its evaluation returns a value. This value is used in instructions.

Expressions are composed of operators, page 678, operands, page 746, and/or assignments. An operand can be a constant, a variable, a function call, or another expression.

Examples

`33`	Constant, literal
`ivar`	Variable
`fct(a,b,c)`	Function call
`a AND b`	Expression
`(x*y) / z`	Expression
`real_var2 := int_var;`	Assignment (also refer to Assignment as Expression, page 361)

Order of Operations

The evaluation of an expression is performed by processing the operators according to certain rules. The operator with the highest order of operation is processed first, then the operator with the next operating level, and so on, until all operators have been processed.

Below you find a table of the ST operators in the order of their ordinal operating level:

Operation	Symbol	Operating Level
placed in parentheses	(expression)	highest order
function call	function name (parameter list)	.....
exponentiation	EXPT	.....
negate	–	.....
building of complements	NOT	.....
multiply	*	.....
divide	/	.....
modulo	MOD	.....
add	+	.....
subtract	–	.....
compare V	<,>,<=,>=	.....
equal to	=	....
not equal to	<>	...
boolean AND	AND	..
boolean XOR	XOR	.
boolean OR	OR	lowest order

Assignment as Expression

As an extension to the IEC 61131-3 standard (ExST), assignments can be used as an expression.

Examples:

`int_var1 := int_var2 := int_var3 + 9;`	(* `int_var1` and `int_var2` both equal to the value of `int_var3` + 9*)
`real_var1 := real_var2 := int_var;`	(* correct assignments, `real_var1` and `real_var2` will get the value of `int_var` *)
`int_var := real_var1 := int_var;`	(* a message will be displayed due to type mismatch `real-int` *)

`IF b := (i = 1) THEN`
`i := i + 1;`
`END_IF`
	(*Expression used inside of IF condition statement: First `b` will be assigned TRUE or FALSE, depending on whether `i` is 1 or not, then the result value of `b` will be evaluated.*)

Instructions

Overview

Instructions describe what to do with the given expressions, page 360.

The following instructions can be used in ST:

Instruction	Example
assignment, page 363	`A:=B; CV := CV + 1; C:=SIN(X);`
Calling a function block, page 364 and using the function block output	`CMD_TMR(IN := %IX5, PT := 300);` `A:=CMD_TMR.Q`
RETURN, page 364	`RETURN;`

IF, page 365	`D:=B*B;`
	`IF D<0.0 THEN`
	`C:=A;`
	`ELSIF D=0.0 THEN`
	`C:=B;`
	`ELSE`
	`C:=D;`
	`END_IF;`

CASE, page 366	`CASE INT1 OF`
	`1: BOOL1 := TRUE;`
	`2: BOOL2 := TRUE;`
	`ELSE`
	`BOOL1 := FALSE;`
	`BOOL2 := FALSE;`
	`END_CASE;`

FOR, page 367	`J:=101;`
	`FOR I:=1 TO 100 BY 2 DO`
	`IF ARR[I] = 70 THEN`
	`J:=I;`
	`'EXIT;`
	`END_IF;`
	`END_FOR;`

WHILE, page 368	`J:=1;`
	`WHILE J<= 100 AND ARR[J] <> 70 DO`
	`J:=J+2;`
	`END_WHILE;`

REPEAT, page 369	`J:=-1;`
	`REPEAT`
	`J:=J+2;`
	`UNTIL J= 101 OR ARR[J] = 70`
	`END_REPEAT;`

EXIT, page 369	`EXIT;`
CONTINUE, page 369	`CONTINUE;`

JMP, page 370	`label1: i:=i+1;`
	`IF i=10 THEN`
	`JMP label2;`
	`END_IF`
	`JMP label1;`
	`label2:`

empty instruction	`;`

Assignment Operators

Default Assignment

On the left side of an assignment, there is an operand (variable, address) to which the value of the expression on the right side is assigned by the assignment operator `:=`.

Also refer to the description of the `MOVE` operator, page 686 which has the same function.

For an example with assignment, refer to the *Example of a Call with Assignments* paragraph, page 195 of the *Calling a Function Block* chapter.

Example
```
a:= b;
```

`a` gets the value of `b`.

Set operator `S=`

The value will be set: if it is once set to TRUE, it will remain TRUE.

Example
```
a S= b;
```

`a` is set to TRUE if `b` is TRUE while processing the assignment.

If `b` is FALSE while processing the assignment, the value of `a` will not be modified.

Reset operator `R=`

The value will be reset: if it is once set to FALSE, it will remain FALSE.

Example
```
a R= b;
```

`a` is set to FALSE as soon as `b` = TRUE.

> **NOTE:** In case of a multiple assignment, set and reset assignments refer to the last member of the assignment.

Example
```
a S= b R= fun1(par1,par2)
```

In this case, `b` will be the reset output value of `fun1`. But `a` does not get the set value of `b`, but gets the set output value of `fun1`.

Consider that an assignment can be used as an expression, page 360. This is an extension to the IEC 61131-3 standard.

Assignment operator `REF`

This operator generates a reference to a value.

Syntax
```
<variable name> REF= <variablename> ;
```

Example
```
refA : REFERENCE TO DUT;
B : DUT;
C : DUT;
A REF= B; // corresponds to A := ADR(B);
A := C; // corresponds to A^ := C;
```

Calling Function Blocks in ST

A function block (abbreviated by FB) is called in structured text according to the following syntax:

<name of FB instance>(FB input variable:=<value or address>|, <further FB input variable:=<value or address>|...further FB input variables);

Example

In the following example, a timer function block (`TON`) is called with assignments for the parameters `IN` and `PT`. Then result variable `Q` is assigned to variable `A`. The timer function block is instantiated by `TMR:TON;`. The result variable, as in IL, is addressed according to syntax <FB instance name>.<FB variable>:

```
TMR(IN := %IX5, PT := 300);
A:=TMR.Q;
```

There is also another syntax available for outputs:

```
fb(in1:=myvar, out1=>myvar2);
```

`RETURN` Instruction

You can use the `RETURN` instruction to leave a POU.

Syntax

```
RETURN;
```

Example

```
IF b=TRUE THEN
RETURN;
END_IF;
a:=a+1;
```

If `b` is TRUE, instruction `a:=a+1;` will not be executed. As a result, the POU will be left immediately.

IF Instruction

With the IF instruction you can test for a condition, and, depending upon this condition, execute instructions.

Syntax

IF <boolean_expression1> THEN

<IF_instructions>

{ELSIF <boolean_expression2> THEN

<ELSIF_instructions1>

..

..

ELSIF <boolean_expression n> THEN

<ELSIF_instructions-1>

ELSE

<ELSE_instructions>}

END_IF;

The segment in brackets {} is optional.

If the <boolean_expression1> returns TRUE, then only the <IF_instructions> are executed and, as a result, none of the other instructions. Otherwise, the boolean expressions, beginning with <boolean_expression2>, are evaluated one after the other until 1 of the expressions returns TRUE. Then only those instructions after this boolean expression and before the next ELSE or ELSIF are evaluated. If none of the boolean expressions produce TRUE, then only the <ELSE_instructions> are evaluated.

Example

```
IF temp<17
THEN heating_on := TRUE;
ELSE heating_on := FALSE;
END_IF;
```

Here, the heating is turned on when the temperature sinks below 17 degrees. Otherwise, it remains off.

CASE Instruction

With the CASE instruction, you can combine several conditioned instructions with the same condition variable in one construct.

Syntax

CASE <Var1> OF

<value1>: <instruction 1>

<value2>: <instruction 2>

<value3, value4, value5>: <instruction 3>

<value6..value10>: <instruction4>

..

..

<value n>: <instruction n>

ELSE <ELSE instruction>

END_CASE;

A CASE instruction is processed according to the following model:

- If the variable in <Var1> has the value <Value I>, then the instruction <Instruction I> will be executed.
- If <Var 1> has none of the indicated values, then the <ELSE Instruction> will be executed.
- If the same instruction is to be executed for several values of the variables, then you can write these values one after the other separated by commas and thus condition the common execution.
- If the same instruction is to be executed for a value range of a variable, you can write the initial value and the end value separated by 2 dots. Therefore, you can condition the common condition.

Example

```
CASE INT1 OF
1, 5: BOOL1 := TRUE;
      BOOL3 := FALSE;
2: BOOL2 := FALSE;
      BOOL3 := TRUE;
10..20: BOOL1 := TRUE;
      BOOL3:= TRUE;
ELSE
    BOOL1 := NOT BOOL1;
    BOOL2 := BOOL1 OR BOOL2;
END_CASE;
```

FOR Loop

With the FOR loop, you can program repeated processes.

Syntax

INT_Var:INT;

FOR <INT_Var> := <INIT_VALUE> TO <END_VALUE> {BY <step size>} DO

<instructions>

END_FOR;

The segment in brackets {} is optional.

The <instructions> are executed as long as the counter <INT_Var> is not greater than the <END_VALUE>. This is checked before executing the <instructions> so that the <instructions> are not executed if <INIT_VALUE> is greater than <END_VALUE>.

When <instructions> are executed, <INT_Var> is increased by <Step size>. The step size can have any integer value. If it is missing, then it is set to 1. The loop will terminate when <INT_Var> is greater than the <END_VALUE>.

Example

```
FOR Counter:=1 TO 5 BY 1 DO
Var1:=Var1*2;
END_FOR;
Erg:=Var1;
```

Assuming that the default setting for Var1 is 1. Then it will have the value 32 after the FOR loop.

> **NOTE:** If <END_VALUE> is equal to the limit value for the data type of <INT_Var> (Counter in the above example), you will produce an infinite, or endless, loop. If Counter is of type SINT, for example, and the <END_VALUE> is 127 (the maximum positive value for a SINT type variable), then the loop can never terminate because adding 1 to this maximum value would result in the variable becoming negative and never exceeding the limits imposed by the FOR instruction.

⚠WARNING

ENDLESS LOOP

Ensure that the variable type used in FOR instructions is of a sufficient capacity to account for the <END_VALUE> + 1.

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

You can use the CONTINUE instruction within a FOR loop. This is an extension to the IEC 61131-3 standard.

WHILE Loop

An alternative to the FOR loop is the WHILE loop, which executes the loop if, and for as long as, a boolean condition is, and remains, TRUE. If the condition is not initially TRUE, the loop is not executed. If the condition which was initially TRUE becomes FALSE, the loop is terminated.

Syntax

WHILE <boolean expression> DO

<instructions>

END_WHILE;

Evidently, the initial and ongoing boolean expression must assume a value of FALSE at some point within the instructions of the loop. Otherwise, the loop will not terminate, resulting in an infinite, or endless, loop condition.

⚠WARNING

ENDLESS LOOP

Ensure that the WHILE loop will be terminated within the instructions of the loop by creating a FALSE condition of the boolean expression controlling the WHILE loop.

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

The following is an example of instructions in the loop causing the loop to terminate:

```
WHILE Counter>0 DO
 Var1 := Var1*2;
 Counter := Counter-1;
END_WHILE
```

You can use the CONTINUE instruction within a WHILE loop.

REPEAT Loop

The REPEAT loop is another alternative to the FOR loop, as it is for the WHILE loop. The REPEAT loop differs from the WHILE loop in that the exit condition is evaluated only after the loop has been executed at least once, at the end of the loop.

Syntax

REPEAT

<instructions>

UNTIL <boolean expression>

END_REPEAT;

The <instructions> are carried out repeatedly as long as the <boolean expression> returns TRUE. If <boolean expression> is produced already at the first UNTIL evaluation, then <instructions> are executed only once. The <boolean expression> must assume a value of TRUE at some point within the instructions of the loop. Otherwise, the loop will not terminate, resulting in an infinite, or endless, loop condition.

⚠WARNING

ENDLESS LOOP

Ensure that the REPEAT loop will be terminated within the instructions of the loop by creating a TRUE condition of the boolean expression.

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

The following is an example of instructions in the loop causing the loop to terminate:

```
REPEAT
 Var1 := Var1*2;
 Counter := Counter-1;
UNTIL
 Counter<=0
END_REPEAT;
```

You can use the CONTINUE instruction within a REPEAT loop. This is an extension to the IEC 61131-3 standard.

CONTINUE Instruction

As an extension to the IEC 61131-3 standard, the CONTINUE instruction is supported within FOR, WHILE, and REPEAT loops. CONTINUE makes the execution proceed with the next loop cycle.

Example

```
FOR Counter:=1 TO 5 BY 1 DO
INT1:=INT1/2;
IF INT1=0 THEN
CONTINUE; (* to avoid division by zero *)
END_IF
Var1:=Var1/INT1; (* only executed, if INT1 is not "0" *)
END_FOR;
Erg:=Var1;
```

EXIT Instruction

The EXIT instruction terminates the FOR, WHILE or REPEAT loop in which it resides without regard to any condition.

JMP Instruction

You can use the JMP instruction for an unconditional jump to a code line marked by a jump label.

Syntax

JMP <label>;

The <label> is an arbitrary, but unique identifier that is placed at the beginning of a program line. The instruction JMP has to be followed by the indication of the jump destination that has to equal a predefined label.

⚠WARNING

ENDLESS LOOP

Ensure that the use of the JMP instruction is conditional.

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

The following is an example of instructions in the create logical conditions that avoid an infinite, or endless, loop between the jump and its destination:

```
aaa:=0;
_label1: aaa:=aaa+1;
(*instructions*)
IF (aaa < 10) THEN
JMP _label1;
END_IF;
```

As long as the variable i, being initialized with 0, has a value less than 10, the jump instruction of the example above will affect a repeated flyback to the program line defined by label _label1. Therefore, it will affect a repeated processing of the instructions comprised between the label and the JMP instruction. Since these instructions also include the increment of the variable i, the jump condition will be infringed (at the ninth check) and program flow will proceed.

You can also achieve this functionality by using a WHILE or REPEAT loop in the example. Generally, using jump instructions reduces the readability of the code.

Comments in ST

You can place comments everywhere within the declaration or implementation part of the ST editor.

There are two possibilities to write comments in a structured text object:

- Start the comment with `(*` and close it with `*)`. This allows you to insert comments which run over several lines.

  Example: `(*This is a comment.*)`

- Use single-line comments as an extension to the IEC 61131-3 standard which are marked in two different ways:
  - `//` denotes the start of a comment that ends with the end of the line.

    Example: `// This is a comment.`
  - `///` denotes the start of a comment that ends with the end of the line.

    Example: `/// This is a comment.`

  Example of comments for tooltips and POU documentation:

  ```
  A tooltip in the header of a POU is defined by the
  following comment:
      // tooltip text - line 1
      // tooltip text - line 2
      // tooltip text - line 3

      Afterwards the documentation is defined as follows:
      /// reStructuredText
  ```

  For further information on handling the two types of comments when creating library documentation using the CODESYS LibDoc Scripting Collection tool with the *LibDocContent* property, refer to the online help under **Software > Libraries, Templates and Examples > Libraries > CODESYS Libraries > Library Development Summary > Concepts and Elements > Library Properties**.

Nested comments: You can place comments within other comments.

Example:

```
(*
a:=inst.out; (* to be checked *)
b:=b+1;
*)
```

In this example, the comment that begins with the first bracket is not closed by the bracket following `checked`, but only by the last bracket.

Object Editors

What’s in This Part

Declaration Editors 373
Device Type Manager (DTM) Editor 380
Data Unit Type (DUT) Editor 381
Global Variables List (GVL) Editor 382
Motion Design Object Editor 383
Network Variables List (NVL) Editor 386
Task Editor 404
Watch List Editor 423
Tools Within Logic Editors 426

Declaration Editors

What's in This Chapter

Textual Declaration Editor 373
Tabular Declaration Editor 374
Declaration Editor in Online Mode 378

Overview

The textual declaration editor serves to create the declaration part of a POU object. It can be supplemented by a tabular view. Any modification made in one of the views is immediately applied to the other one.

Depending on the current settings in the declaration editor options, either only the textual or only the tabular view will be available. You can toggle between both via buttons (**Textual** / **Tabular**) at the right border of the editor window.

Usually, the declaration editor is used in combination with the programming language editors. Therefore, it will be placed in the upper part of the window which opens when you are going to edit or view (monitor) an object in offline or online mode. The declaration header describes the POU type (for example: PROGRAM, FUNCTION_BLOCK, FUNCTION). It can be extended by POU-global pragma attributes.

The online mode of the declaration editor, page 378 is structured like that of a **Watch** view.

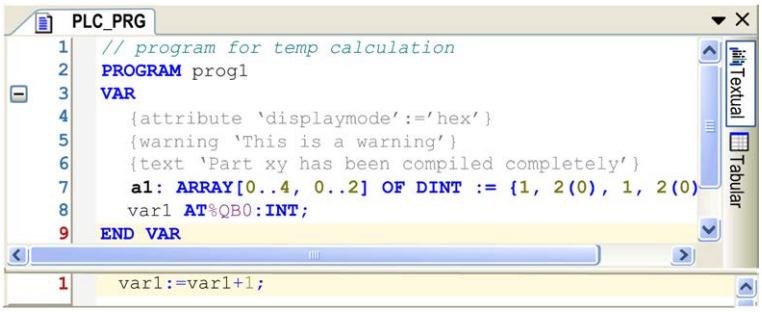
Variable declaration is also performed in **Global Variable Lists** and **Data Unit Types**, which are created in separate editors.

Also refer to the *Variables Declaration* chapter, page 550.

Textual Declaration Editor

Overview

Textual editor view



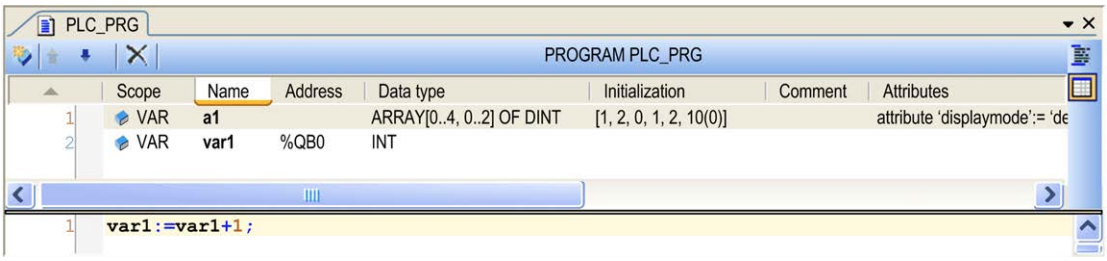
Behavior and appearance are determined by the respective current text editor settings in the **Options** and **Customize** dialog boxes. There you can define the default settings for highlight coloring, line numbers, tabs, indenting, and many more options. Editing functions, such as copy and paste, are available. Block selection is possible by pressing the **Alt** key while selecting the desired text area with the mouse.

- To scroll within the editor view, hold down the **Shift** key and use the scroll wheel of the mouse.
- To zoom within the editor view, hold down the **Ctrl** key and use the scroll wheel of the mouse.

Tabular Declaration Editor

Overview

Tabular editor view



The tabular view of the editor provides columns for the usual definitions for variable declaration, page 550: **Scope**, **Name**, **Address**, **Data type**, **Initialization**, **Comment** and (pragma) **Attributes**. The particular declarations are inserted as numbered lines.

To add a new line of declaration above an existing one, first select this line and execute the command **Insert** from the toolbar or the contextual menu.

To add a new declaration at the end of the table, click beyond the last existing declaration line and also use the **Insert** command.

The newly inserted declaration by default first uses scope **VAR** and the recently entered data type. The input field for the obligatory variable **Name** opens automatically. Enter a valid identifier and close the field by pressing the **Enter** key or by clicking another part of the view.

Double-click a table cell to open the respective possibilities to enter a value.

Double-click the **Scope** to open a list from which you can choose the desired scope and scope attribute keyword (flag).

Type in the **Data type** directly or click the **>** button to use the **Input Assistant** or the **Array** wizard.

Type in the **Initialization** value directly or click the **...** button to open the **Initialization value** dialog box, page 376. This is useful especially in case of arrays and structured variables.

Each variable is declared in a separate line where the lines are numbered.

You can change the order of lines (line numbers) by selecting a line and move it one up or down by the **Move up** or **Move down** command from the toolbar or the contextual menu.

You can sort the list of declarations according to each of the columns by clicking the header of the respective column.

To delete one or several declarations, select the respective lines and press the **Delete** key or execute the **Delete** command from the contextual menu or click the button in the toolbar.

Declaration of Arrays

For the declaration of array variables, use the arrow button **>** at the right side of the **Type** field and select **Array Wizard**. The **Array** dialog box opens.

Fill at least the fields marked with a red exclamation mark icon. Define the **Dimension**s by entering the lower and upper limits, and the **Base Type** of the variable. You can click the arrow button to open the **Input Assistant** dialog box or another **Array Wizard** for declaring the base type.

You can define an array of variable length with `[*,*,*]`. Arrays of variable length can only be used in *VAR_IN_OUT* declarations of function blocks, methods, and functions. To declare an array of variable length, enter an asterisk `*` for each dimension. This results in `ARRAY [*..*] OF INT`. After you have confirmed with **OK**, adapt the dimension string to `[*]` (one asterisk only).

Example for a two-dimensional array of variable length:
```
ARRAY [*,*]
```

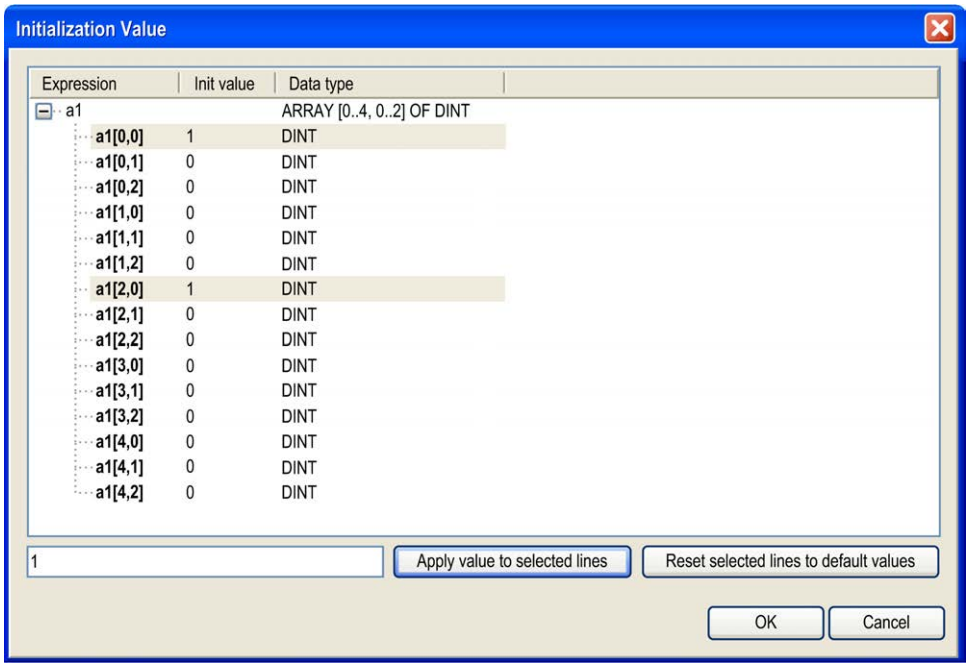
The **Result** area of the dialog box provides a preview of the configured array declaration.

For further information, refer to the *Arrays* description, page 651.

Click **OK** to close the declaration dialog box. The variable declaration appears in the declaration editor in accordance to the IEC syntax.

Initialization Value

Initialization value dialog box



The **Expressions** of the variable are displayed with the present initialization values. Select the desired variables and edit the initialization value in the field below the listing. Then click the **Apply value to selected lines** button. To restore the default initializations, click the **Reset selected lines to default values** button.

Press **Ctrl** + **Enter** to insert line breaks in the **Comment** entry.

If the variable to be initialized is a function block instance with an extended `FB_Init` method, page 579, an additional table is displayed that lists the `FB_Init` parameters. The following differences apply to this table in contrast to the **Initialization Value** table:

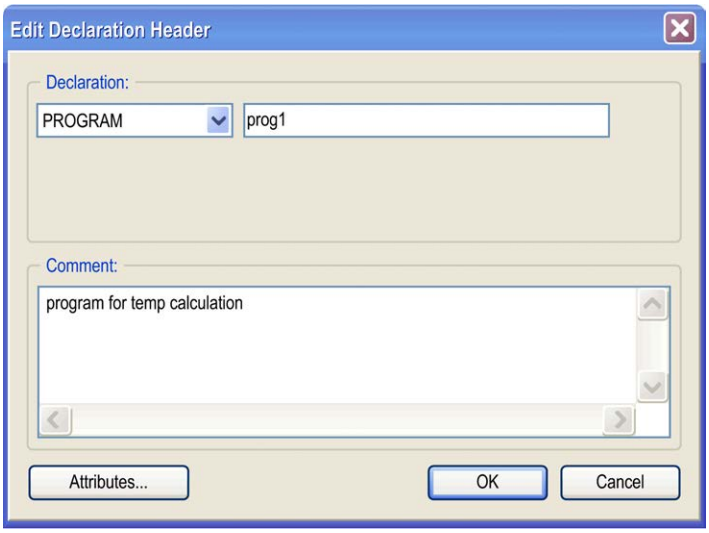
- Assign an initialization value to each variable. Otherwise, the **OK** button remains disabled.
- Complex data types (structures, arrays) cannot be expanded and the components they contain cannot be displayed. Use a corresponding variable to initialize complex data types.

`FB_Init` parameters are marked in the **Auto Declare** dialog box by a corresponding symbol that is displayed after the initialization value.

Edit Declaration Header

You can edit the declaration header in the **Edit Declaration Header** dialog box. Open it by clicking the header bar of the editor (PROGRAM PLC_PRG in the figure above) or via the command **Edit Declaration Header**.

Edit Declaration Header dialog box



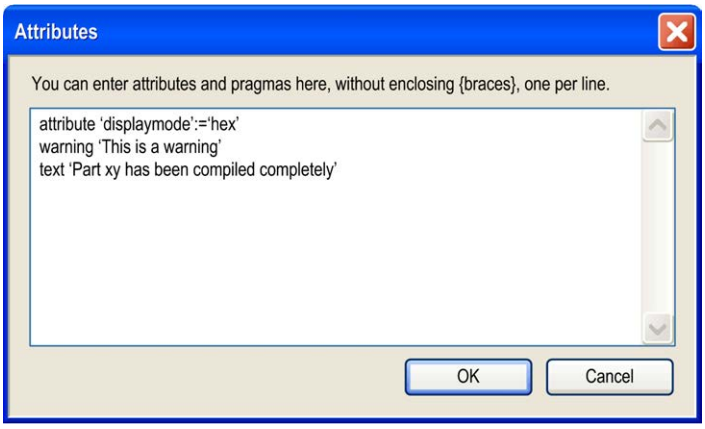
The **Edit Declaration Header** dialog box provides the following elements:

Element	Description
Declaration	Insert type (from the selection list) and name of the POU object.
Comment	Insert a comment. Press **Ctrl** + **Enter** to insert line breaks.
Attributes	Opens the **Attributes** dialog box (see further below in this chapter) for inserting pragmas and attributes.

Attributes

In the **Edit Declaration Header** dialog box, click the **Attributes...** button to open the **Attibutes** dialog box. It allows you to enter multiple attributes and pragmas in text format. Insert them without enclosing {} braces, use a separate line per each. For the example shown in the following image, see the corresponding textual view above in the graphic of the textual editor view, page 373.

Attributes dialog box



Declaration Editor in Online Mode

Overview

The online view of the declaration editor presents a table similar to that used in watch views, page 424. The header line shows the actual object path <*device name*>.<*application name*>.<*object name*>. The table for each watch expression shows the type and present value as well as - if currently set - a prepared value for forcing or writing. If available, a directly assigned IEC **Address** and / or **Comment** are displayed in further columns.

The **Value** column displays the value on the controller offering monitoring functionality. If the variable is an array with more than 1,000 elements, you can define the range of the array indices to monitor. For further information, refer to Defining the Range of the Array Indices, page 379.

To establish a prepared value for a variable, either use the **Prepare Value** dialog box or click in the assigned field of the column **Prepared value** and directly type in the desired value. In case of enumerations, a list showing the enumeration values opens to select a value. In case of a boolean variable, the handling is even easier.

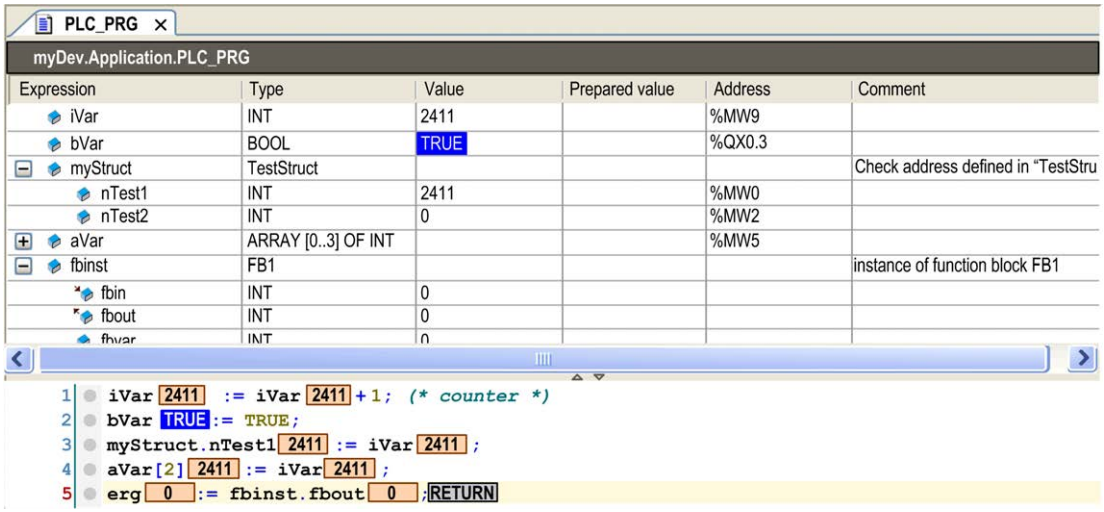
You can toggle boolean preparation values by use of the **Return** or **Space** key according to the following order:

- If the value is TRUE, the preparation steps are FALSE -> TRUE -> nothing.
- If the value is FALSE, the preparation steps are TRUE -> FALSE -> nothing.

If a watch expression (variable) is a structured type, for example, an instance of a function block or an array variable, then a plus or minus sign precedes the expression. With a mouse-click on this sign the particular elements of the instanced object can be additionally displayed (see `fbinst` in the following image) or hidden (see `aVar`). Icons indicate whether the variable is an input, output or an ordinary variable.

When you point with the cursor on a variable in the implementation part, a tooltip shows the declaration and comment of the variable. See the following image showing the declaration editor in the upper part of a program object `PLC_PRG` in online view:

Online view of the declaration editor



Defining the Range of the Array Indices

If the variable is an array with more than 1,000 elements, you can define the range of the array indices to monitor as follows:

Step	Action
1	Double-click the **Type** column. **Result**: The **Monitoring Area** dialog box opens. The declared array range is specified as the **Valid area** for monitoring. A maximum of 20,000 elements can be monitored per array.
2	Enter a **Start** index.
3	Enter an **End** index for the array to be monitored.

The toggle button on the right-hand side of the scrollbars allows you to couple and uncouple the scrollbars.

- : The scrollbars are coupled. You can move this area while maintaining the same size.
- : The scrollbars are not coupled. You can increase or decrease the size of the area to be monitored.

Device Type Manager (DTM) Editor

What's in This Chapter

DTM Editor 380

DTM Editor

Overview

The DTM editor view depends on the device type manager.

For further information on DTMs, refer to the *Device Type Manager (DTM) User Guide* (see Device Type Manager (DTM), User Guide).

For a list of the DTM versions currently supported, refer to the Release Notes of your installation.

Data Unit Type (DUT) Editor

What's in This Chapter

Data Unit Type Editor 381

Data Unit Type Editor

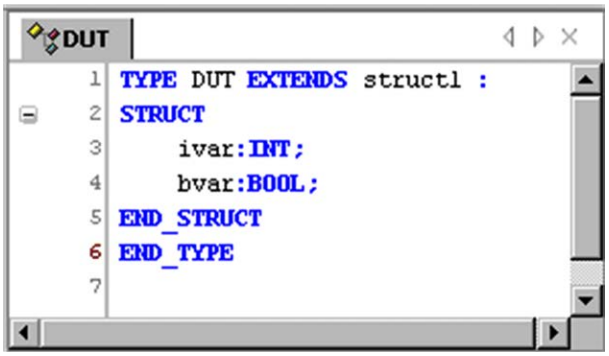
Overview

You can create user-defined data types, page 635 in the Data Unit Type editor (DUT editor). This is a text editor and behaves according to the currently set text editor options.

The DUT editor will be opened automatically in a window when you add a DUT object in the **Add object** dialog box. In this case, it provides by default the syntax of an extended structure declaration. You can use it as desired to enter a simple structure declaration or to enter the declaration of another data type unit, for example an enumeration.

The editor also opens when you open an existing DUT object currently selected in the POUs view.

DUT editor window



Global Variables List (GVL) Editor

What's in This Chapter

GVL Editor..... 382

GVL Editor

Overview

The GVL editor is a **Declaration Editor** for editing **Global Variables Lists**. The GVL editor works as does the Declaration Editor and corresponds to the options, both offine and online, set for the text editor. The declaration starts with `VAR_GLOBAL` and ends with `END_VAR`. These keywords are provided automatically. Enter valid declarations of global variables between them.

GVL editor



GVL Editor Used as Persistent Variables Editor

The GVL editor can also be used to manage **Persistent Variables** objects. In this case, the editor contains declarations of `VAR_GLOBAL PERSISTENT` variables. For further information, refer to the chapter *Persistent Variables*, page 218 .

Motion Design Object Editor

What's in This Chapter

Motion Design Object Editor..... 383

Motion Design Object Editor

Overview

The motion design object editor is exclusive to the M262 Motion Controller

The motion design object editor provides a generic base for configuring different motion design objects such as:

- A real axis as, for example, the Lexium 32S servo drive.
- A variable that implements a motion design objects interface.

To add a motion design object editor to your project, right-click the **Application** node, and execute the command **Add object > Motion Design Object...**. The **Add Motion Design Object** dialog box allows you to select the type of motion design object to be created.

Adding a Motion Design Object of Type Virtual Axis

For the **Type** = **Virtual Axis**, the following elements are available in the **Add Motion Design Object** dialog box:

Element	Description
Mode section	The **Mode** settings you specify here can be modified in the **Configuration** tab of the motion design object editor.
Target instance	Click the browse button (**...**) to select one of the supported function blocks or enter it directly into the field on the left-hand side.
Device object	Select a device object from the controller configuration that implements one of the interfaces of a motion design object, such as a Lexium 32 S standalone servo drive.
Generate global instance	A global variable of a motion design object interface is created. It can be accessed by IEC code by using the name of the object.
OK button	Click the **Add** button to: • Create a **Motion Design Object** node as a subnode of the selected **Application** node. • Open the motion design object editor for the virtual axis.

Motion Design Object Editor of a Virtual Axis

The motion design object editor allows you to edit the parameter values. For the **Type** = **Virtual Axis**, it provides two tabs with the following parameters.

Elements of the **Parameterisation** tab:

Element	Description
Axis configuration section	The **Axis configuration** section allows you to select the graphical representation of the **Axis position** on the lower left side of the editor: • The option **Modulo axis** displays the **Axis position** as a modulo control. The **Modulo Period** parameter defines the value that is assigned to one revolution of the circle in the diagram. • The option **Linear axis** displays the **Axis position** as a linear control. Activate the **Soft limits** option to define a range of values for the x axis of the graph. If the axis is not within the defined boundaries, the axis position is displayed red.
Errorstop ramp configuration section	
Deceleration	Enter a value for the deceleration ramp in Units/s^2.
Jerk	Enter a value for the jerk ramp in Units/s^2.
Axis position	Graphical control of the position of the axis in modulo or linear mode, as defined with the **Axis configuration** parameter.
Axis State	Present state of the axis represented a a graphical chart.
Motion components	List of the components used for the virtual axis, indicating the **Position**, **Velocity**, and **Acceleration** values.

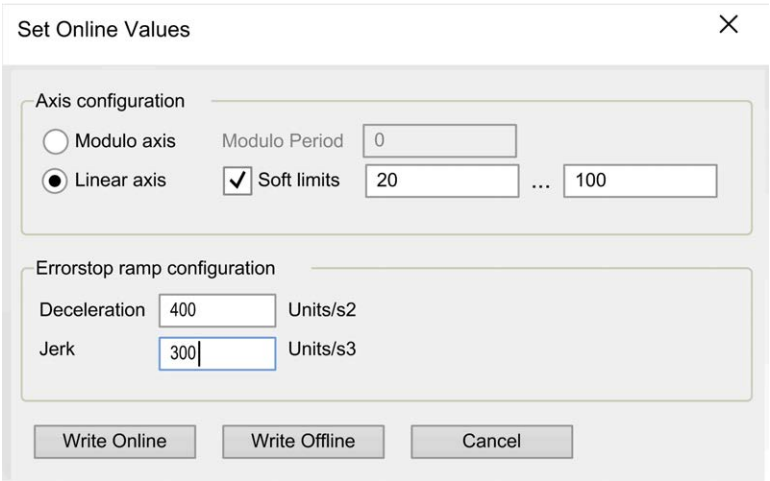
Elements of the **Configuration** tab:

The **Configuration** tab contains the **Mode** and **Access specifier** parameters that are similar to the **Add Motion Design Object** dialog box, page 383 and that can be modified here.

Modifying the Configuration in Online Mode

You can modify the values of the parameters if you are logged in to the controller by double-clicking an editable cell of the parameter list in the **Configuration** tab.

Result: The **Set Online Values** dialog box opens. It displays the configuration that is present in the controller.



You can modify the configuration according to your individual requirements.

The following options are available:

- Click the **Write Online** button to write the values to the controller. If errors are detected during this operation, a message box is displayed. After the values have been successfully written to the controller, the dialog box is closed.
- Click the **Write Offline** button to save the values to the project. The dialog box is closed.

Network Variables List (NVL) Editor

What's in This Chapter

Information on the NVL Editor 386
General Information on Network Variables 387

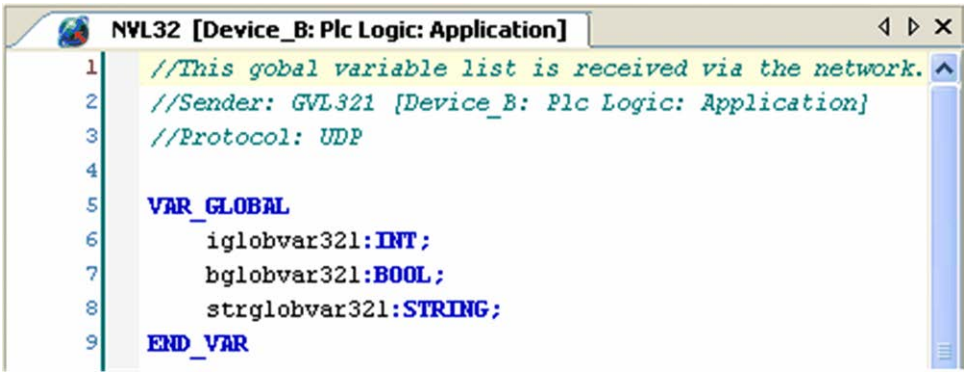
Information on the NVL Editor

Network Variables List Editor

Overview

The NVL editor is a **Declaration Editor** for editing **Network Variables Lists**. The NVL editor works as does the Declaration Editor and corresponds to the options, both offine and online, set for the text editor. The declaration starts with `VAR_GLOBAL` and ends with `END_VAR`.These keywords are provided automatically. Enter valid variable declarations, page 550 of global variables between them.

NVL editor



General Information on Network Variables

Introduction to Network Variables List (NVL)

Overview

The Network Variables List (NVL) feature consists of a fixed list of variables that can be sent or received through a communication network. This enables data exchange within a network via network variables, if supported by the controller (target system).

The list must be defined in the sender and in the receiver controllers (and can be handled in a single or in multiple projects). Their values are transmitted via broadcasting through User Datagram Protocol (UDP) datagrams. UDP is a connectionless Internet communications protocol defined by IETF RFC 768. This protocol facilitates the direct transmission of datagrams on Internet Protocol (IP) networks. UDP/IP messages do not expect a response, and are therefore ideal for applications in which dropped packets do not require retransmission (such as streaming video and networks that demand real-time performance).

The NVL functionality is a powerful feature. It allows you to share and monitor data between controllers and their applications. However, there are no restrictions as to the purpose of the data exchanged between controllers, including, but not limited to, attempting machine or process interlocking or even controller state changes.

> **NOTE:** The type of the network variable is not shared between different controllers. You have to ensure that the used types have the same definition on all devices; otherwise NVL communication is not possible. This applies, for example, to the types *SEC.ETH_R_STRUCT* or *SEC.PLC_R_STRUCT*. They are available by default in various controllers with different size or fields.

Only you, the application designer and/or programmer, can be aware of all the conditions and factors present during operation of the machine or process and, therefore, only you can determine the proper communication strategies, interlocks and related safeties necessary for your purposes in exchanging data between controllers using this feature. Strict care must be taken to monitor this type of communication feature, and to be sure that the design of the machine or process will not present safety risks to people or property.

⚠WARNING

LOSS OF CONTROL

- Perform a Failure Mode and Effects Analysis (FMEA), or equivalent risk analysis, of your application, and apply preventive and detective controls before implementation.
- Provide a fallback state for undesired control events or sequences.
- Provide separate or redundant control paths wherever required.
- Supply appropriate parameters, particularly for limits.
- Review the implications of transmission delays and take actions to mitigate them.
- Review the implications of communication link interruptions and take actions to mitigate them.
- Provide independent paths for control functions (for example, emergency stop, over-limit conditions, and error conditions) according to your risk assessment, and applicable codes and regulations.
- Apply local accident prevention and safety regulations and guidelines.[1]
- Test each implementation of a system for proper operation before placing it into service.

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

[1] For additional information, refer to NEMA ICS 1.1 (latest edition), *Safety Guidelines for the Application, Installation, and Maintenance of Solid State Control* and to NEMA ICS 7.1 (latest edition), *Safety Standards for Construction and Guide for Selection, Installation and Operation of Adjustable-Speed Drive Systems* or their equivalent governing your particular location.

You can use Diagnostic (see Network Variable Configuration, SE_NetVarUdp Library Guide) and Error Management (see Network Variable Configuration, SE_NetVarUdp Library Guide) function blocks as well as network properties parameters to monitor the health, status and integrity of communications using this feature. This feature was designed for data sharing and monitoring and cannot be used for critical control functions.

Network Variables List (NVL)

The network variables to be exchanged are defined in the following two types of lists:

- Network Variables Lists (NVL sender) in a sending controller
- Network Variables List (NVL receiver) in a receiving controller

The corresponding NVL (sender) and NVL (receiver) contain the same variable declarations. You can view their contents in the respective editor that opens after double-clicking the NVL (sender) or NVL (receiver) node in the **Devices tree**.

An NVL (sender) contains the network variables of a sender. In the **Network properties** of the sender, protocol and transmission parameters are defined. According to these settings, the variable values are broadcasted within the network. They can be received by the controllers that have a corresponding NVL (receiver).

> **NOTE:** For network variables exchange, the respective network libraries must be installed. This is done automatically for the network type UDP as soon as the network properties for an NVL (sender) are set.

Network variables are broadcasted from the NVL (sender) to one or more NVL (receiver). For each controller, you can define NVL (sender) as well as NVL (receiver). Thus each controller can act as sender as well as receiver.

An NVL (sender) can be provided by the same or by another project. So, when creating an NVL (receiver), the NVL (sender) can either be chosen from a selection list of all available NVL (sender) within the network, or it can be read from an export file, which previously has been generated (for example, by using the **Link to File** dialog box) from the NVL (sender).

> **NOTE:** An export file is needed if the NVL (sender) to be used is defined within another project.

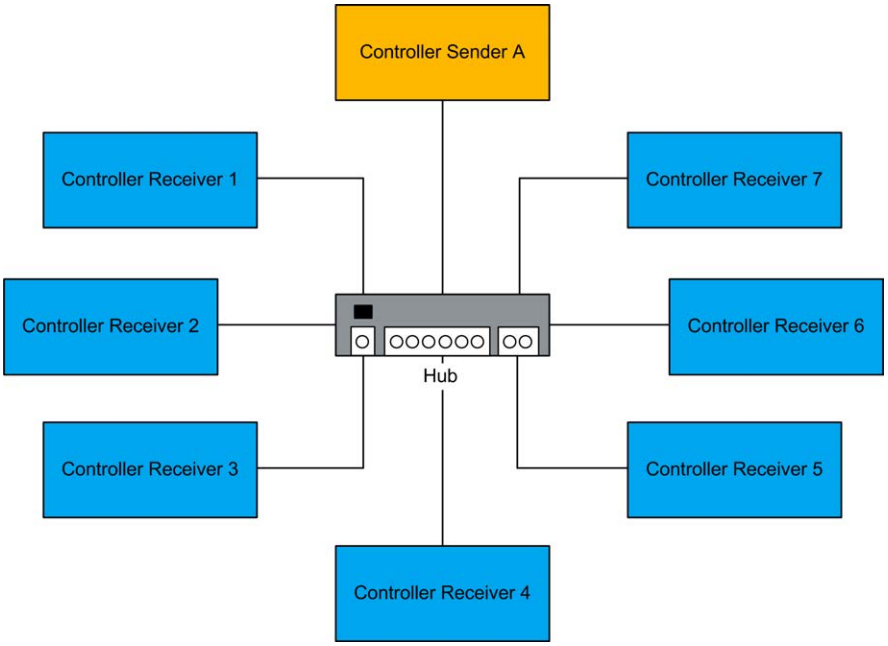
NVL Considerations

The following table shows the list of controllers that support the network variables list (NVL) functionality:

Function Name	M241 M251	M262 Logic / Motion	LMC Eco LMC Pro LMC Pro2
Network Variables List	Yes	Yes	Yes

NOTE: Since the exchange of network variables is performed via UDP, adjust the firewall settings of your controller accordingly. For further information, refer to the How to Configure the Firewall for PacDrive LMC Controllers User Guide and to the Programming Guide of your controller.

The figure shows a network consisting of one sender and the maximum of seven receivers:



Controller Sender A: Sender with the NVL (sender) and receiver controller with network variables lists (NVL (receiver))

Controller Receiver 1...7: Receivers (with NVL (receiver)) from A and sender controller (NVL (sender)) only for A

Configuring the Network Variables Exchange

Overview

To exchange network variables between a sender and a receiver, one sender and one receiver controller must be available in the **Devices tree**. These are the controllers that are assigned the network properties described below.

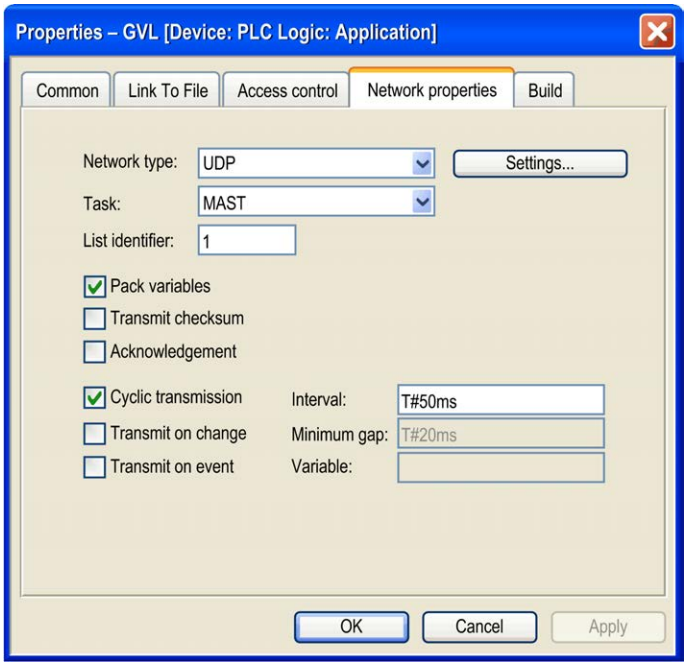
Proceed as follows to configure the network variables list:

Step	Action
1	Create a sender and a receiver controller in the **Devices tree**.
2	Create a program (POU) for the sender and receiver controller.
3	Add a task for the sender and receiver controller. **NOTE:** In order to maintain performance transparency, you should set the task priority of the dedicated NVL task to something greater than 25, and regulate communications to avoid saturating the network unnecessarily.
4	Define the NVL (sender) for the sender.
5	Define the NVL (receiver) for the receiver.

An example with further information is provided in the Appendix (see Network Variable Configuration, SE_NetVarUdp Library Guide).

Global Variables List

To create the NVL (sender), define the following network properties in the **GVL > Properties > Network properties** dialog box:



Description of parameters

Parameter	Default Value	Description
Network type	**UDP**	Only the network type **UDP** is available. To change the **Broadcast Address** and the **Port**, click the **Settings...** button.
Task	**MAST**	Select the task you configured below the **Task Configuration** item for executing NVL code. To help maintain performance transparency, it is a good practice configure a cycle time **Interval** ≥50 ms for this task. **NOTE:** In order to maintain performance transparency, you should set the task priority of the dedicated NVL task to something greater than 25, and regulate communications to avoid saturating the network unnecessarily. **NOTE:** NVL must not be linked to the **MAST** M241/M251 controller.
List identifier	**1**	Enter a unique number for each NVL (sender) on the network. It is used by the receivers for identifying the variables list (see Network Variable Configuration, SE_NetVarUdp Library Guide).
Pack variables	activated	With this option activated, the variables are bundled in packets (datagrams) for transmission. If this option is deactivated, one packet per variable is transmitted.
Transmit checksum	deactivated	Activate this option to add a checksum to each packet of variables during transmission. Receivers will then check the checksum of each packet they receive and will reject those with a non-matching checksum. A notification will be issued with the `NetVarError_CHECKSUM` parameter (see Network Variable Configuration, SE_NetVarUdp Library Guide).
Acknowledgement	deactivated	Activate this option to prompt the receiver to send an acknowledgement message for each data packet it receives. A notification will be issued with the `NetVarError_ACKNOWLEDGE` parameter (see Network Variable Configuration, SE_NetVarUdp Library Guide) if the sender does not receive this acknowledgement message from the receiver before it sends the next data packet.
Cyclic transmission • **Interval**	activated	Select this option for cyclic data transmission at the defined **Interval**. This **Interval** should be a multiple of the cycle time you defined in the task for executing NVL code to achieve a precise transmission time of the network variables.

Parameter	Default Value	Description
Transmit on change • **Minimum gap**	deactivated • **T#20ms**	Select this option to transmit variables whenever their values have changed. **NOTE:** After the first download or using of **Reset Cold** or **Reset Warm** command in Online Mode the receiver controllers are not updated and keep their last value, whereas the sender controller value becomes 0 (zero). The **Minimum gap** parameter defines a minimum time span that has to elapse between the data transfer.
Transmit on event • **Variable**	deactivated • –	Select this option to transmit variables as long as the specified **Variable** equals TRUE. The variable is checked with every cycle of the task for executing NVL code.

Description of the button **Settings...**

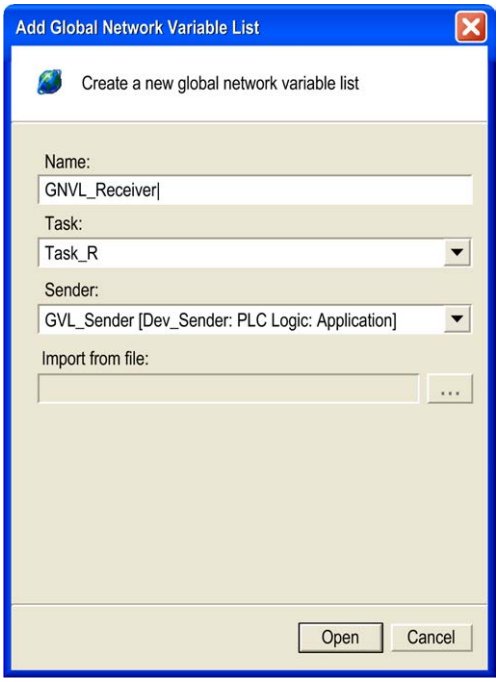
Parameter	Default Value	Description
Port	1202	Enter a unique port number (≥ 1202) for each NVL (sender).
Broadcast Address	255.255.255.255	Enter a specific broadcast IP address for your application.

Network Variables List (Receiver)

A global network variables list can only be added in the **Devices tree**. It defines variables, which are specified as network variables in another controller within the network.

Thus, an NVL (receiver) object can only be added to an application if an NVL (sender) with network properties (network variables list) has already been created in one of the other network controllers. These controllers may be in the same or different projects.

To create the NVL (receiver), define the following parameters in the **Add Object > Global Network Variable List** dialog box:



Description of parameters

Parameter	Default Value	Description
Name	**NVL**	Enter a name for the NVL (receiver).
Task	task defined in the **Task Configuration** node of this **Application**	Select a task from the list of tasks which will receive the frames from the sender that are available under the **Task Configuration** node of the receiver controller.
Sender	1 of the NVL (sender) available in the project	Select the NVL (sender) from the list of the NVL (sender) with network properties available in the project. Select the entry **Import from file** from the list to use an NVL (sender) from another project. This activates the **Import from file:** parameter below.
Import from file:	–	This parameter is only available after you selected the option **Import from file** for the parameter **Sender**. The **...** opens a Windows Explorer window that allows you to browse to the export file **.gvl* you created from an NVL (sender) in another project. For further information, refer to the *How to Add an NVL (Receiver) From a Different Project* paragraph below.

How to Add an NVL (Receiver) in the Same Project

When you add an NVL (receiver) via the **Add Object** dialog box, the corresponding NVL (sender) that are found within the present project for the present network are provided for selection in the **Sender** list box. NVL (sender) from other projects must be imported (see the *How to Add an NVL (Receiver) From a Different Project* paragraph below).

Due to this selection, each NVL (receiver) in the present controller (sender) is linked to one specific NVL (sender) in another controller (receiver).

Additionally, you have to define a name and a task, that is responsible for handling the network variables, when adding the NVL (receiver).

How to Add an NVL (Receiver) From a Different Project

Alternatively to directly choosing an NVL (sender) from another controller, you can also specify an NVL (sender) export file you had generated previously from the NVL (sender) by using the **Link to file** properties. This allows you to use an NVL (sender) that is defined in another project.

To achieve this, select the option **Import from file** for the **Sender:** parameter and specify the path in the **Import from file:** parameter.

You can modify the settings later on via the **Properties - GVL** dialog box.

NVL (Receiver) Properties

If you double-click an NVL (receiver) item in the **Devices tree**, its content will be displayed on the right-hand side in an editor. But the content of the NVL (receiver) cannot be edited, because it is only a reference to the content of the corresponding NVL (sender). The exact name and the path of the sender that contains the corresponding NVL (sender) is indicated at the top of the editor pane together with the type of network protocol used. If the corresponding NVL (sender) is changed, the content of the NVL (receiver) is updated accordingly.

Network Variables List (NVL) Rules

Rules on the Amount of Data

Due to some performance limitations, respect the following rules:

Number	Rule
1	Data transmission from one NVL (sender) to one NVL (receiver) depends on the controller. • For Modicon controllers, a maximum of 200 bytes per variable is allowed. • For PacDrive controllers, a maximum of 255 bytes per variable is allowed.
2	• For Modicon controllers, data exchange between several NVL (sender) of one controller and their associated NVL (receiver) should not exceed 1000 bytes for all variables. • For PacDrive controllers, the number of bytes for all network variables is not limited.

Rules on the Number of Datagrams for Modicon Controllers

To limit the maximum cycle time of NVL tasks for Modicon controllers, respect the following rules:

Number	Rule	Description
1	Limit the number of received datagrams per cycle to 20.	When the limit is exceeded, the remaining datagrams are treated in the next cycle and a notification **Received overflow** is raised in the diagnostics data (see Network Variable Configuration, SE_NetVarUdp Library Guide). One datagram can contain up to 256 bytes. That means that you should not exceed the limit of 5120 bytes of data transmitted to one receiver.
2	Limit the number of transmitted datagrams per cycle to 20.	When the limit is exceeded, the remaining datagrams are treated in the next cycle. A notification **Transmit overflow** is raised in the diagnostics data (see Network Variable Configuration, SE_NetVarUdp Library Guide). One datagram can contain up to 256 bytes. That means that you should not exceed the limit of 5120 bytes of data transmitted by one sender controller.

If the number of received / transmitted datagrams per cycle exceeds the limit several times, the following may happen:

- Loss of UDP datagrams
- Incoherent or inconsistent exchange of variables

Adapt the following parameters according to your needs:

- Cycle time of sender controller
- Cycle time of receiver controller
- Number of senders in the network

NOTICE

LOSS OF DATA

Thoroughly test your application for proper transmission and reception of UDP datagrams prior to placing your system into service.

Failure to follow these instructions can result in equipment damage.

Maximum Number of NVL (Sender)

Define a maximum of seven NVL (sender) per controller (sender) to help maintain performance transparency.

Rules on Task Cycle Times of NVL (Sender) and NVL (Receiver)

To help avoid reception overflow, you must define a cycle time for the task that manages the NVL (sender) transmission that is at least two times greater than the cycle time of the task that manages the NVL (receiver) reception.

Rules on the List Identifier Protection

The NVL function includes a list identifier checking:

The list identifier helps to avoid that an NVL (sender) from two separate controllers with the same list identifier (see dialog box **GVL > Properties > List identifier:**) sends datagrams to the same NVL (receiver) of any controller. If the **List Identifier** is not unique, this can cause an interruption in the exchange of variables.

NOTICE
LOSS OF COMMUNICATION Ensure that the list identifier in the network is only used by one IP address. **Failure to follow these instructions can result in equipment damage.**

The list identifier checking function is implemented in the receiver controller.

If an NVL (receiver) detects that two different IP addresses are using the same list identifier, the receiver immediately stops to receive datagrams.

Furthermore, a notification is issued in the `NETVARGETDIAGINFO` function block. The IP addresses of the two senders are provided in the output parameters `dwDuplicateListIdIp1` and `dwDuplicateListIdIp2` of this function block (see Network Variable Configuration, SE_NetVarUdp Library Guide).

With the function block `NETVARRESETERROR`, the detected NVL errors are reset and the communication is restarted.

Consistency in the Type of Network Variables

NOTE: The type of the network variable is not shared between different controllers. You have to ensure that the used types have the same definition on all devices; otherwise NVL communication is not possible.

This applies, for example, to the types *SEC.ETH_R_STRUCT* or *SEC.PLC_R_STRUCT*. They are available by default in various controllers with different size or fields.

Operating State of the Sender and the Receiver

Operating State

Operating State of the...		**Network Variables Behavior**
Sender	**Receiver**	
RUN	RUN	Network variables are exchanged between the sender and the receiver.
STOP	RUN	The sender is no longer sending variables to the receiver. The network variables are not exchanged between sender and receivers.
RUN	STOP	The receiver is not processing network variables from the sender. When the receiver returns to RUN state, the network variables are again processed by the receiver.
STOP	STOP	No variables are exchanged.

NOTE: Several communication initialization errors (`NetVarError_INITCOMM`) are detected, when you change the operating state of the sender from STOP to RUN.

Events in the Task that Manages NVL

If the following events occur in the task that manages NVL, the behavior expected for the NVL is the same as if the controller was in STOP state in the array above:

- an exception occurs in the application which suspends the task
- a breakpoint is hit or a single cycle is processed on the task

Example

Overview

In the following example, a simple network variables exchange is established. In the sender controller, an NVL (sender) is created. In the receiver controller, the corresponding NVL (receiver) is created.

Perform the following preparations in a standard project, where a sender controller **Dev_Sender** and a receiver controller **Dev_Receiver** are available in the **Devices tree**:

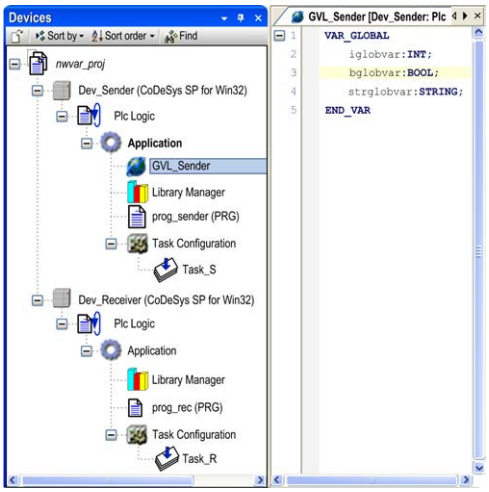
- Create a POU (program) **prog_sender** below the **Application** node of **Dev_Sender**.
- Under the **Task Configuration** node of this application, add the task **Task_S** that calls **prog_sender**.
- Create a POU (program) **prog_rec** below the **Application** node of **Dev_Receiver**.
- Under the **Task Configuration** node of this application, add the task **Task_R** that calls **prog_rec**.

  NOTE: The two controllers must be configured in the same subnet of the Ethernet network.

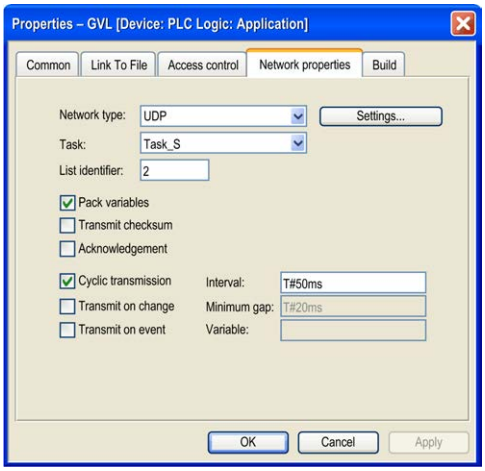
Defining the NVL (Sender)

Step 1: Define a global variable list in the sender controller:

Step	Action	Comment
1	In the **Devices tree**, right-click the **Application** node of the controller **Dev_Sender** and execute the command **Add Object > Global Variable List...**.	The **Add Global Variable List** dialog box is displayed.
2	Enter the **Name** `NVL_Sender` and click **Open** to create a new global variable list.	The **NVL_Sender** node appears below the **Application** node in the **Devices tree** and the editor is opened on the right-hand side.

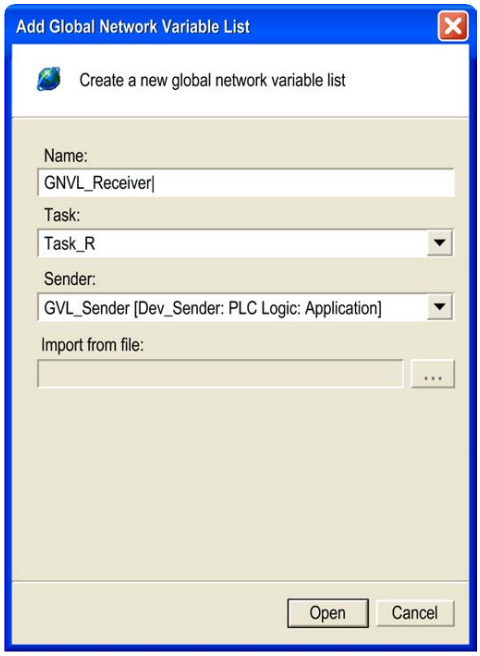
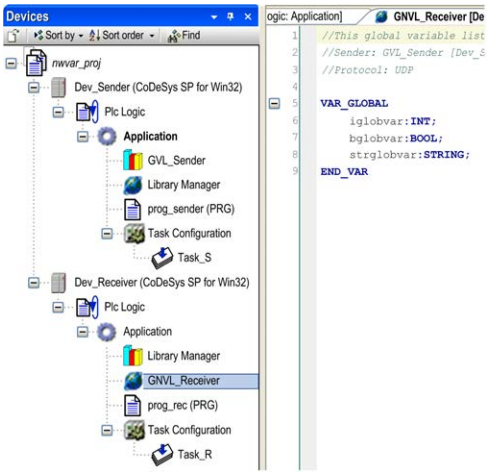
3	In the editor on the right-hand side, enter the following variable definitions:
	`VAR_GLOBAL`
	`iglobvar:INT;`
	`bglobvar:BOOL;`
	`strglobvar:STRING;`
	`END_VAR`
	
		–

Step 2: Define the network properties of the NVL (sender):

Step	Action	Comment
1	In the **Devices tree**, right-click the **NVL_Sender** node and execute the command **Properties...**.	The **Properties - NVL_Sender** dialog box is displayed.
2	Open the **Network properties** tab and configure the parameters as shown in the graphic: 	–
3	Click **OK**.	The dialog box is closed and the NVL (sender) network properties are set.

Defining the NVL (Receiver)

Step 1: Define a global network variable list in the receiver controller:

Step	Action	Comment
1	In the **Devices tree**, right-click the **Application** node of the controller **Dev_Receiver** and execute the command **Add Object > Global Network Variable List...**.	The **Add Global Network Variable List** dialog box is displayed.
2	Configure the parameters as shown in the graphic. 	This global network variable list is the counterpart of the NVL (sender) defined for the sender controller.
3	Click **Open**.	The dialog box is closed and the **GNVL_Receiver** appears below the **Application** node of the **Dev_Receiver** controller:  This NVL (receiver) automatically contains the same variable declarations as the **GVL_Sender**.

Step 2: View and / or modify the network settings of the NVL (receiver):

Step	Action	Comment
1	In the **Devices tree**, right-click the **GNVL_Receiver** node and execute the command **Properties...**.	The **Properties - GNVL_Receiver** dialog box is displayed.
2	Open the **Network settings** tab.	–

Step 3: Test the network variables exchange in online mode:

Step	Action	Comment
1	Under the **Application** node of the controller **Dev_Sender**, double-click the POU **prog_sender**.	The editor for **prog_sender** is opened on the right-hand side.
2	Enter the following code for the variable `iglobvar`: 	–
3	Under the **Application** node of the controller **Dev_Receiver**, double-click the POU **prog_rec**.	The editor for **prog_rec** is opened on the right-hand side.
4	Enter the following code for the variable `ivar_local`: 	–
5	Log on with sender and receiver applications within the same network and start the applications.	The variable `ivar_local` in the receiver gets the values of `iglobvar` as currently shown in the sender.

Compatibility

Introduction

Even if the controllers work with applications of different versions of the programming system (for example, V2.3 and V3.x), communication via network variables is possible.

However, the file formats of the different export files between versions (*.exp versus *.gvl) makes it impossible to simply import and export these files between projects.

If an NVL (receiver) is set up in the latest version (for example, V3.x), the required network parameters configuration must be provided by a sender of the latest version (for example, V3.x). An export file **.exp* created from a sender by an earlier version (for example, V2.3) does not contain this information.

A solution for exchanging network variables between applications of different programming system versions is provided in the following paragraphs.

Updating the Global Network Variables List

To exchange network variables between applications of different programming system versions (for example, V2.3 and V3.x), update the global network variables list by performing the following steps:

Step	Action	Comment
1	Re-create the network variables list (NVL) that is already available in the earlier version (V2.3) in the latest version (V3.x).	To achieve this, add an NVL (sender) with network properties, containing the same variables declarations as in the NVL of the earlier version (V2.3).
2	Export the new NVL (sender) to an *.*exp* file by using the **Link to File** tab.	**NOTE:** Activate the option **Exclude from build** in the **Build** tab to keep the NVL (sender) in the project to help prevent precompile events and ambiguous names. Deactivate the option to re-create the *.*exp* file again in case any modifications on the NVL (sender) are required.
3	Re-import the list.	To achieve this, create a new NVL (receiver) by using the previously generated *.*exp* file in order to create an appropriately configured receiver list.

These steps are illustrated by the following example.

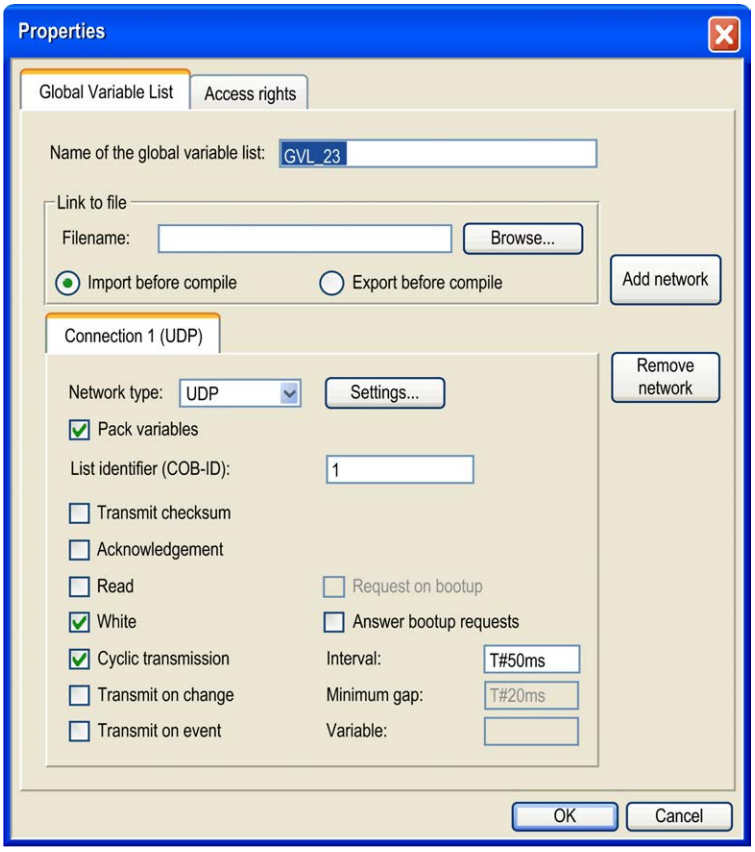
Example

In this example, the variable `trans23`, that is defined in a V2.3 application, is made available for a later version (V3.x).

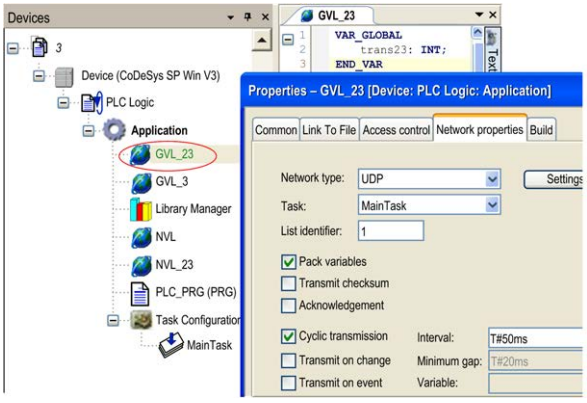
The following conditions are defined:

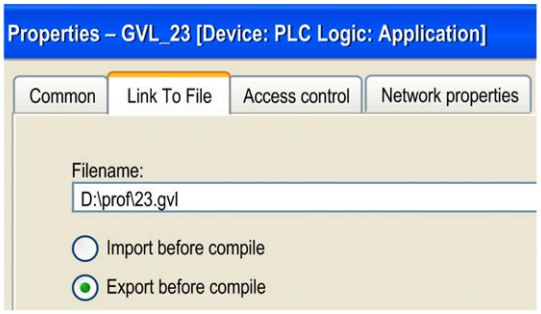
Condition	Description

1	In the earlier programming system version (V2.3) the project *23.pro* contains a global variables list **GVL_23** with the following declaration:
	`VAR_GLOBAL`
	`trans23:INT;`
	`END_VAR`

2	The network properties of **GVL_23** are configured as follows:
	
	
	NOTE: The export of this **GVL_23** creates a **.exp* file, that only contains the following variable declaration:
	`VAR_GLOBAL`
	`trans23:INT;`
	`END_VAR`
	The **.exp* file does not contain any configuration settings.

The following table shows the next steps to be executed for re-creating **GVL_23** in the latest version (V3.x):

Step	Action	Comment
1	Add an NVL (sender) object named **GVL_23** to an application.	–
2	Set the network properties as defined within the *23.pro* project.	

Step	Action	Comment
3	In the **Link to File** tab, configure a target export file *23.gvl*.	
4	In the **Build** tab, set the **Exclude from Build** option.	This setting allows you to keep the file on disk for later modifications. 

5	Compile the project.	The *23.gvl* file is generated and contains variable and configuration settings:
		`<GVL>`
		`<Declarations><![CDATA[VAR_GLOBAL trans23: INT; END_VAR]]></Declarations>`
		`<NetvarSettings Protocol="UDP">`
		`<ListIdentifier>1</ListIdentifier>`
		`<Pack>True</Pack>`
		`<Checksum>False</Checksum>`
		`<Acknowledge>False</Acknowledge>`
		`<CyclicTransmission>True</CyclicTransmission>`
		`<TransmissionOnChange>False</TransmissionOnChange>`
		`<TransmissionOnEvent>False</TransmissionOnEvent>`
		`<Interval>T#50ms</Interval>`
		`<MinGap>T#20ms</MinGap>`
		`<EventVariable>`
		`</EventVariable>`
		`<ProtocolSettings>`
		`<ProtocolSetting Name="Port" Value="1202" />`
		`<ProtocolSetting Name="Broadcast Adr." Value="192.168.101.167" />`
		`</ProtocolSettings>`
		`</NetvarSettings>`
		`</GVL>`

6	Add an NVL (receiver) object in the V3.x project from the *23.gvl* export file (with the **Import from file:** command).	This serves to read the variable `trans23` from the controller of the earlier programming system (V.2.3). If both, the project from the earlier version (V2.3) as well as the application from the latest version (V3.x), are running within the network, the application from the latest version (V3.x) can read variable `trans23` from project *23.pro*.

Task Editor

What's in This Chapter

Information on the Task Configuration 404
Task Groups Tab 405
Properties Tab 406
System Events Tab 407
Monitor Tab 409
Variable Usage Tab 410
Stack Usage Tab 411
Configuration of a Specific Task 413
Task Processing in Online Mode 416
Multicore 417
Multicore - Tools 420
CPU Load Tab 422

Information on the Task Configuration

Overview

The task configuration defines one or several tasks for controlling the processing of an application program. Thus, task configuration is an essential object for an application and must be available in the **Applications Tree**.

Description of the Task Configuration Tree

At the topmost position of a task configuration tree, there is the entry **Task Configuration** . Below there are the defined tasks, each represented by the task name. The POU calls of the particular tasks are displayed in the task configuration tree.

You can edit the task tree (add, copy, paste, or remove tasks) by the corresponding commands usable for the **Applications tree**. For example, for adding a new task, select the **Task Configuration** node, click the green plus button, and execute the command **Task...**. Alternatively, you can right-click the **Task Configuration** node, and execute the command **Add Object > Task...**.

Configure the particular tasks in the task editor, page 413 which additionally provides a monitoring view in online mode. The options available for task configuration depend on the controller platform.

Task configuration in **Applications tree**



Tasks

A task, page 413 is used to control the processing of an IEC program. It is defined by a name, a priority and by a type determining which condition will trigger the start of the task. You can define this condition by a time (cyclic, freewheeling) or by an internal or external event which will trigger the task; for example, the rising edge of a global project variable or an interrupt event of the controller.

For each task, you can specify a series of program POUs that will be started by the task. If the task is executed in the present cycle, these programs will be processed for the length of 1 cycle.

The combination of priority and condition will determine in which chronological order, page 416 the tasks will be executed.

For each task, you can configure a time control (watchdog). The possible settings depend on the specific controller platform.

Keep in mind that all tasks share the same process image.

When you develop a project with several tasks, ensure that the input as well as the output data are copied to a location where they can only be accessed by one task to help avoid concurrent access to the same data by different tasks.

To help avoid consistency and synchronization issues caused by, for example, multiple read and write operations to one variable or when accessing other global objects (global variables or POUs), use function blocks of the SysSem library, for example.

Task Groups Tab

Overview

The **Task Groups** tab of the **Task Configuration** allows you to define task groups that can be distributed over the individual processor cores in Multicore, page 417 controllers. Use the **Core** parameter to define the strategy for binding the task of a task group to processor cores.

NOTE: This tab is exclusive to Multicore controllers.

Elements of the Tab

The **Task Groups** tab provides the following elements:

Element	Description
Add Group button	Click to create a new task group with the default name `NewGroup_<no>`.
Remove Group button	Click to delete the selected task group.
Group Name field	Double-click the field to edit the name of the group.
Core field	Select one option from the list: • **Free floating** As a prerequisite for this option, use multicore operators, such as `TEST_AND_SET`, page 745 or `__COMPARE_AND_SWAP`, page 743. The IEC tasks of the task group are dynamically distributed to the individual processor cores by the load balancing function of the operating system. This has the effect that the load is automatically distributed and manual intervention is not possible. • **Sequentially pinned**(1) As a prerequisite for this option, use multicore operators, such as `TEST_AND_SET`, page 745 or `__COMPARE_AND_SWAP`, page 743. The IEC tasks of the task group are permanently bound to different processor cores. If the number of tasks exceeds the number of processor cores, the tasks are bound to the individual processor cores, starting with Core1 and continuing with Core2, Core3, etc. In the event of an overflow, the process is continued with Core0, Core1, etc. Manual intervention is not possible. • **Fixed pinned**(1) The IEC tasks of the task group are permanently bound to a processor core that is assigned by the runtime (by default, Core1). If several task groups with **Fixed pinned** setting are available, each task group is bound to a different processor core starting with Core1. • **<fixed core number>**(1) Enter a processor core number. The IEC tasks of the task group are permanently bound to this processor core. If the processor core does not exist on the controller, an error is detected during the download.
Priority field	Displays the priority of the task as configured with the **Priority** parameter, page 413.

(1): The **Sequentially pinned**, **Fixed pinned**, and **<fixed core number>** settings can help to obtain more deterministic jitter times because these settings are not influenced by the dynamic distribution of tasks to the individual cores at runtime. However, the settings prevent the operating system from distributing the load across the individual processor cores.

NOTE: Test the **Sequentially pinned**, **Fixed pinned**, and **<fixed core number>** settings before usage to verify that they lead to a significantly improved real-time behavior.

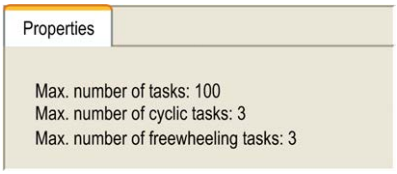
NOTE: For data consistency without using multicore operators, assign a task group/tasks for one machine (for example, motion, inputs/outputs, HMI) to a single core.

Properties Tab

Overview

When the **Task Configuration**, page 404 node is selected, the **Properties** tab will be opened in the **Task Configuration** editor.

Task configuration, **Properties** tab, example



Information on the current task configuration as provided by the controller will be displayed, for example, the maximum allowed numbers of tasks per task type.

System Events Tab

Overview

Use the **System Events** tab of the **Task Configuration** editor to call a project function by a system event (not by a task). You can define the functions that are called per event, and you can activate or deactivate each system event individually.

This function is not available for all supported controllers. Consult the *Programming Guide* specific to your controller for further information.

Description of the Buttons

Button	Description
Add Event Handler...	Opens the **Add Event Handler** dialog box that allows you to define an assignment between an event and the function it calls.
Remove Event Handler	Deletes the selected list assignment.
Event Info	Displays information of the corresponding event library.
Open Event Function	Opens the editor of the new function for the selected assignment. The implementation language of the new function is selected in the **Add Event Handler** dialog box.

Add Event Handler Dialog Box

Element	Description
Event	Displays a list of system events for selection. The available events depend on the target controller. Unavailable events are marked with a red symbol.
Function to call	Enter a name for the function. A new **POU**, type **Function** is created with this name and added to the **Applications tree** when you confirm the **Add Event Handler** dialog box with **OK**.
Scope	• Select the option **Application** to make the new function available for the application. • Select the option **POUs** to make the new function available for the entire project.
Implementation language	Select the programming language for the new function.
Description	Provides a short description of the selected event.

List of System Events

The system events you define with the **Add Event Handler** dialog box are displayed in the list of the **System Events** tab:

Column	Description
Name	Indicates the name of the **Event** you defined in the **Add Event Handler** dialog box.
Description	Indicates a short description of the selected event.
Function to call	Indicates the name of the function you defined with the parameter **Function to call** in the **Add Event Handler** dialog box.
Active	The option is by default selected and the function is activated. To deactivate the function, deselect the option **Active**.

The list of the **System Events** tab displays additional information in online mode. Consult the *Programming Guide* specific to your controller for controller-specific information on the supported system events.

Column	Description
Event Status	**0** indicates that no error has been detected. If the value is not 0, then an error has been detected.
Number of Calls	Indicates the number of times the event has occurred and the associated function has been called.

Button	Description
Online Reset	Click the **Online Reset** button to reinitialize the event lists, and to reset the counter for the events / function calls. Events that are not correctly initialized are displayed with a red **Event Status** cell.

System Events

The table lists the system events that can be displayed:

Event	Description	Task	Debugging
AfterWritingOutputs	Called after writing outputs.	IEC task	Yes
BeforeReadingInputs	Called before reading inputs.	IEC task	Yes
CodeInitDone	The event is sent after new application code is initialized during online-change.	Communication task	No
Exception	The event is sent if an exception has been detected in the context of an application.	During an IEC exception, the event is called from the system exception task or the task with the exception itself.	Depends on the task.
Login	Login of a client in this application.	Communication task	No
Logout	Logout of a client from this application.	Communication task	No
OnlineChangeDone	Called after application online change.	Communication task	No
PrepareOnlineChange	Called before application online change.	Communication task	No
sercos_BeforeChangeToPhase0	Called before change to phase 0.	TskSercos3App	No
sercos_BeforeConnectSlaves	Called before real slaves are mapped to logical slaves in the **Devices tree**.	TskSercos3App	No

Event	Description	Task	Debugging
sercos_TasksPhase0	Called during change to phase 0.	TskSercos3App	No
sercos_TasksPhase1	Called during change to phase 1.	TskSercos3App	No
sercos_TasksPhase2PostFW	Called after change to phase 2	TskSercos3App	No
sercos_TasksPhase2PreFW	Called before change to phase 2	TskSercos3App	No
sercos_TasksPhase3PostFW	Called after change to phase 3	TskSercos3App	No
sercos_TasksPhase3PreFW	Called before change to phase 3	TskSercos3App	No
sercos_TasksPhase4	Called during change to phase 4.	TskSercos3App	No

Monitor Tab

Overview

If it is supported by the target system, the monitoring functionality is allowed. This is a dynamic analysis of the execution time and the number of the calls which are controlled by a task. In online mode, the task processing can be monitored.

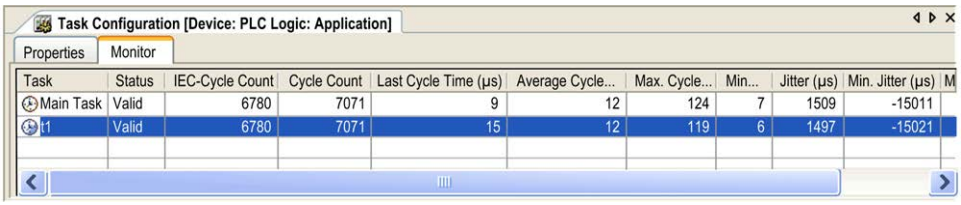
Online View of the Task Configuration Editor

When you select the top node in the **Task Configuration** tree, besides the **Properties** tab, page 406, the **Monitor** tab is available. In online mode, it shows the status and some current statistics on the cycles and cycle times in a table view. The update interval for the values is the same as used for the monitoring of controller values.

Description of the Elements

When the top node in the **Task Configuration** tree is selected, besides the **Properties** dialog, page 406 on a further tab the **Monitoring** dialog is available. In online mode, it shows the status and some current statistics on the cycles and cycle times are displayed in a table view. The update interval for the values is the same as used for the monitoring of controller values.

Task Configuration, **Monitor**



For each task the following information is displayed in a line:

Task	Task name as defined in the **Task configuration**.
State	Possible entries: • **Not created**: has not been started since last update; especially used for event tasks • **Created**: task is known in the runtime system, but is not yet set up for operation • **Valid**: task is in normal operation • **Exception**: task has got an exception
IEC-Cycle Count	Number of run cycles since having started the application; 0 if the function is not supported by the target system.
Cycle Count	Number of already run cycles (depending on the target system, this can be equal to the IEC Cycle Count, or bigger if cycles are even counted when the application is not running.)
Last Cycle Time (µs)	Last measured runtime in µs
Average Cycle Time (µs)	Average runtime of all cycles in µs
Max. Cycle Time (µs)	Maximum measured runtime of all cycles in µs
Min. Cycle Time (µs)	Minimum measured runtime of all cycles in µs
Jitter (µs)	Present value of the periodic jitter[1)] in µs
Min. Jitter (µs)	Minimum measured periodic jitter[1)] in µs
Max. Jitter (µs)	Maximum measured periodic jitter[1)] in µs
[1)] Periodic jitter J^{per} is the deviation of the task cycle time T^{per} from the specified task cycle time T^0. $J_{per} = T_{per} - T_0$ The cycle time T^0 is specified in the configuration of the task as **Interval**.	

To reset the values to 0 for a task, place the cursor on the task name field and execute the **Reset** command available in the contextual menu.

Variable Usage Tab

Overview

The **Variable Usage** tab of the **Task Configuration** provides an overview of the variables and their usage in the tasks.

Variable Usage List

The list of the **Variable Usage** tab provides the following information:

Column	Description
Variables	Indicates the name of the variable.
Type	Indicates the data type of the variable.
Count	Indicates the number of tasks that access the variable.
<task name>	Indicates the access to the variable: • **r** = read • **w** = write • **rw** = read/write

Commands of the Contextual Menu

Right-click an entry in the list to display the contextual menu that allows you to:

- Hide tasks from the list by deselecting the task.
- Display the cross-reference list to variables by executing the command **Browse Cross References**.

Stack Usage Tab

Overview

The **Stack Usage** tab of the **Task Configuration** displays the stack usage per application, per task and for the POUs in the maximum call stack column.

As a prerequisite for data to be displayed in this tab, the application code must have been generated. A code generation run is performed automatically when logging in with the application to the controller. To start code generation explicitly, execute the **Build > Generate Code** command.

Elements of the Tab

The **Stack Usage** tab provides the following elements:

Element	Description
Override default stack size from the target settings option	By default, this check box is cleared and the **Available stack size** value cannot be edited. Select the option to make the **Available stack size** field editable. If you enter a value that is lower than the calculated value, consider the hazard message below this table.
Reset to default button	When the **Override default stack size from the target settings** option is selected, click this button to reset the value of the **Available stack size** field to the default value that is read from the device description.
Available stack size field	By default, this field is read-only and indicates the available stack size (in bytes) that is read from the device description. Example: **4,194,304** To edit this field manually, select the **Override default stack size from the target settings** check box. Consider the hazard message below the table. During compilation, the value entered is verified and an error is detected when the calculated stack size exceeds the value entered.
Assumed stack size for external calls field	This field is read-only and indicates the general estimated value of stack size consumption by an external call (in bytes) that is read from the device description. Example: The value **24** indicates that it is estimated that a call to an external function consumes 24 bytes of the stack. To use another value than the estimated stack size provided by the device description, assign a value of your choice to an external call by using the attribute `'estimated-stack-usage'`, page 606. Consider the hazard message above the table.
Task/Maximum Call Stack column	This column indicates call hierarchy, that is, the call stack that was determined during compile. To populate this view with values, start a code generation by executing the **Build > Generate Code** command.
Stack Size column	The cells of the **Stack Size** column indicate the calculated stack size (in bytes) per called POU or the stack size added up by the call hierarchy for a task. **NOTE:** The figures displayed are calculated depending on estimations. For further information, refer to Calculation of the Stack Usage, page 413. The figures are updated during each compile operation. For POUs displayed in red, a stack overflow has been detected. The POUs that are in the hierarchy below are also displayed in red due to memory overflow.

Manually decreasing the **Available stack size** value may lead to stack overflow exceptions.

⚠WARNING

UNINTENDED EQUIPMENT OPERATION

- Ensure that the manually entered **Available stack size** value does not cause a stack overflow.
- Ensure that the manually entered **Assumed stack size for external calls** value does not cause a stack overflow.
- Do not modify the **Available stack size** parameter or the **Assumed stack size for external calls** attribute without having a clear and specific understanding of the affect of the modification(s) in your application.

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

Calculation of the Stack Usage

The stack usage is calculated statically using the static call stack. The calculation is based on the following principles:

- The calculation starts with the hidden/implicit callee for the entry of the task.
- For each IEC program call (for example, `PLC_PRG`), the stack usage of each IEC POU that is called is calculated recursively. The largest possible stack usage of a POU is added to its own local stack usage in order to determine the maximum stack usage of the POU.
- For dynamic calls, the stack usage is calculated as follows:

  The stack usage of possible implementations is determined and the largest stack usage is assumed. For interface methods, possible implementations and for function block methods, possible overrides are taken into account.
- For external calls, the value specified by the target device is assumed. If no value is specified by the device, then 0 is assumed.
- In the case of recursion, the calculation is aborted. The value of the type pragma attribute `'estimated-stack-usage'`, page 606 is applied for the recursively called code. If no value is specified, then 0 is assumed.

  NOTE: The displayed values are estimations. They may be too high because the assumed deepest call path is never run through at runtime. They may also be too low because the estimations of recursions or external calls were not adequate.

Configuration of a Specific Task

Overview

When you insert a task in the **Task Configuration** node of the **Applications tree**, the task editor view for setting the task configuration opens with the **Configuration** tab.

It also opens if you double-click an available task (for example, **MAST**) in order to modify the configuration of the task.

NOTE: You can modify the task name by editing the respective entry in the **Applications tree**.

Insert the desired attributes.

Priority	
Priority (0...31)	A number from 0...31; 0 is the highest priority, 31 is the lowest The default value for new tasks is defined by your controller. **NOTE:** The assigned priority is only considered when tasks are grouped to one processor core. **NOTE:** Consider the controller-specific task settings to find the settings for your application tasks. This may be important when assigning priority to tasks dedicated to communications, and its relation to topics such as cybersecurity.
Task group	This function is exclusive to Multicore controllers. Assigned task group, page 405 can be assigned to specific processor cores in multicore devices. Example: **IEC-Tasks** In the **Devices tree**, the task group is displayed in brackets at the end of the task node name.

Type	
The target device defines which task types are supported. Not all types are available for some target devices. Consult the *Programming Guide* specific to your controller for more information.	
Cyclic	The task will be processed cyclic according to the time definition (task cycle time) given in the field **Interval** (see below).
Event	The task will be started as soon as the variable defined in the **Event** field gets a rising edge.
Freewheeling	The task will be processed as soon as the program is started and at the end of one run it will automatically be restarted in a continuous loop. There is no cycle time defined.
External	The task will be started as soon as the system event, which is defined in the **External Event** field, occurs. It depends on the target, which events will be supported and offered in the selection list. (Not to be mixed up with system events.)
Status	The task will be started if the variable defined in the **Event** field is TRUE.

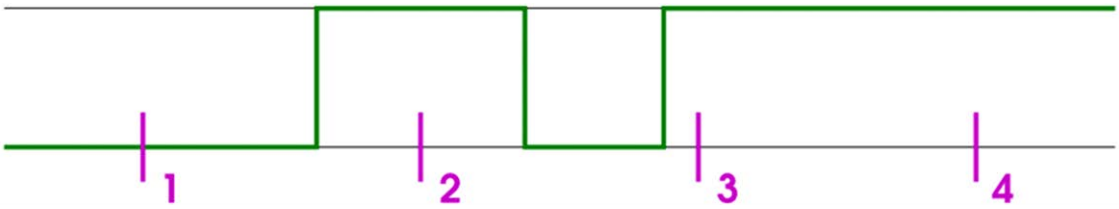
Obligatory Entries Depending on Task Type

Entry	**Description**
Interval (for example, **t#200ms**)	Obligatory for task type **Cyclic**. The time (in milliseconds [ms]) after which the task should be restarted. When you set the task cycle time, consider the bus system used by the application. For example, on a CAN bus system, you can set the **Bus cycle task** in the **CANopen I/O Mapping** tab. The task cycle time must match the transmission rate and the number of frames used on the bus. Additionally, the times set for heartbeat, nodeguarding and sync always should be a multiple of the task cycle time. Otherwise, CAN frames may go unrecognized. For further information, refer to the *Device Editor* part of the online help. Deviations of the task from the configured task cycle time are displayed at runtime as periodic jitter in the **Monitor** tab, page 409
Event	Obligatory for type **Event** or triggered by an **External event**. A global boolean variable which will trigger the start of the task as soon as a rising edge is detected. Use button **...** or the **Input Assistant** to get a list of all available global event variables. **NOTE:** If the event that is driving a task stems from an entry, there must be at least one task which is not driven by events. Otherwise, the I/Os will never get updated and the task will never get started. **NOTE:** Only internal IEC variables and values of onboard touchprobes and digital inputs (controller) are permitted. Referencing a property (including system parameter) in an event task will lead to a watchdog exception error, page 415 being detected during download.

Difference Between Status and Event

The specified event being TRUE fulfills the start condition of a status driven task, whereas an event driven task requires the change of the event from FALSE to TRUE. If the event changes too fast from TRUE to FALSE and back to TRUE, then this event may be left undetected and thus the **Event** task will not be started.

The following example illustrates the resulting behavior of the task in reaction to an event (green line):



At sampling points 1...4 tasks of different types show a different reaction:

Behavior at Point:	**1**	**2**	**3**	**4**
Status	No start	Start	Start	Start
Event	No start	Start	No start because the event changed too fast from TRUE to FALSE and back to TRUE	No start

Watchdog Settings

For each task, you can configure a timeout control (watchdog).

The default watchdog settings depend on your controller.

When the **Enable** option is activated (check mark is set), the watchdog is enabled. When the task watchdog is enabled, an **Exception** error is raised if the execution time of the task exceeds the defined task time limit (**Time**) given in the defined **Sensitivity**.

When the exception is detected, the application is stopped. If option **Update IO while in stop** is enabled in the controller settings dialog box, the outputs are set to the pre-defined default values depending upon the particular controller platform.

> **NOTE:** This function is not available for all supported controllers. Consult the *Programming Guide* specific to your controller.

The defined **Sensitivity** is taken into account when determining when to raise an exception error. The **Sensitivity** allows you to adjust for variations in cycle times in the execution of the task. The **Sensitivity** can be defined as follows:

- After consecutive timeouts:
  - **Sensitivity** set to 0 or 1: exception in the first cycle after time expires
  - **Sensitivity** set to 2: exception in the second cycle after time expires
  - **Sensitivity** set to n: exception in the n^{th} cycle after time expires
- After a single timeout: exception if the cycle time of the present cycle is longer than (task time limit * sensitivity).

> **NOTE:** The watchdog function is not available in simulation mode.

Consult the *Programming Guide* specific to your controller, chapter *System and Task Watchdog*, for information on task time, sensitivity and other possible watchdog parameters.

POUs

The POUs which are controlled by the task are listed here in a table with the POU name and an optional **Comment**. Above the table there are commands for editing:

- In order to define a new POU, open the **Input Assistant** dialog box via the command **Add Call**. Choose 1 of the programs available in the project. You can also add POUs of type program to the list by drag and drop from the **Applications tree**.
- In order to replace a program call by another 1, select the entry in the table, open the **Input Assistant** via command **Change Call..** and choose another program.
- In order to delete a call, select it in the table and use the command **Remove Call**.
- The command **Open POU** opens the selected program in the corresponding editor.

The sequence of the listed POU calls from top to bottom determines the sequence of execution in online mode. You can shift the selected entry within the list via the commands **Move up** and **Move down**.

Task Processing in Online Mode

Task Scheduling

The processing of the POU (of type program) calls will be done according to their order (top down) in the **Task Editor**. If a POU is called which is available with the same name both in the **Applications tree** assigned to the application, as well as in a library or project-globally in the **Global** node of the **Applications tree**, the one will be executed that is directly declared below in the **Applications tree**.
Consult the *Programming Guide* specific to your controller for more information on task scheduling.

Multicore

Overview

This function is exclusive to Multicore controllers. Consult the *Programming Guide* specific to your controller for further information about whether the controller supports multicore.

If an EcoStruxure Machine Expert application with multiple tasks is executed on a multicore controller without using the multicore feature, the distribution of tasks is automatically performed by the scheduler of the operating system and no manual intervention is possible.

The multicore feature allows you to assign IEC tasks to dedicated processor cores, taking the acquired number of processor cores into account. This can result in improved performance. As a prerequisite, it is a good practice to split the IEC program into several tasks.

Two different strategies are supported for distributing IEC tasks to different processor cores:

- Each IEC task is linked and fixed to a specific processor core.

  The task is executed on one specific processor core. By grouping tasks, multiple tasks can be linked to the same processor core. This is useful, for example, if the IEC program cannot run on multiple processor cores simultaneously.

- IEC tasks are executed on all available processor cores.

  The operating system distributes one or more tasks of a group to the processor cores.

  NOTE: For data consistency without using multicore operators, assign a task group/tasks for one machine (for example, motion, inputs/outputs, HMI) to a single core.

Considerations Concerning Distributing Tasks over Multiple Processor Cores

Consider the following specific behavior of the IEC program when distributing IEC tasks to different processor cores.

- Task processing according to the priority is not available. The assigned priority is only considered when tasks are grouped to one processor core.
- Cycle consistency of data in the IEC task with the highest priority is not available. To use consistent values within one cycle, data must be copied locally at the beginning of the IEC task cycle.
- A shared process image is provided to all tasks. The (internal) `ReadInputs()` function is called at the start of each task and the `WriteOutputs()` function is called at the end. These functions read from and write to the shared process image. Thus, the Input Processing Output (IPO) model is valid for each individual task.

  When the bus cycle task is called, the shared process image is physically written and the packets are transmitted. The bus cycle task itself waits for each call of `WriteOutputs()` and `ReadInputs()` of the tasks. This helps to ensure data consistency within the bus cycle task. Configure the bus cycle task in the **PLC settings** view, page 128.

  - Outputs can only be assigned to one task. To display the I/O access, open the **Task Deployment** view of the device editor, page 143.
  - Inputs can be assigned to multiple tasks, but direct access is not allowed. Atomic operations are required to write the inputs to local variables of the task.
  - A second task that updates the process image with `ReadInputs()` in parallel to the first task also updates the inputs of the first task while it is running.
  - Data consistency of a task can be ensured only on single cores for the task with the highest priority.
- For consistent counters (incrementer, decrementer), use the atomic external library function *SysCpuAtomicAdd()* (refer to the SysCpuHandling.library).

Considerations Concerning Data Consistency

Concerning data consistency, consider the following.

- Bit access (data type BIT) is not processed consistently (atomically) on multicore controllers in the IEC program. It is a good practice to use the external library function *SysCpuTestAndSetBit()* (refer to the SysCpuHandling.library).
- Simple data types up to a width of 32 bits (such as BOOL, BYTE, WORD/INT, DWORD/DINT) are processed consistently (atomically) in the IEC program on multicore controllers.
- Data types with 64 bits (LINT, LWORD, and LREAL) are processed consistently (atomically) in the IEC program exclusively on 64-bit systems and multicore systems.
- For accessing complex data types (STRING, FB, STRUCT, and ARRAY), use specific measures, such as copying data locally, using a **Global Variable List (tasklocal)** or using semaphores to synchronize the data and keep consistency.
- In the **Variable Usage** tab of the task configuration, page 410, you can verify whether a variable in an IEC task has read or write access.
- On multicore systems, a memory reordering effect can occur. You can use the IEC operator `__MemoryBarrier()`.

How to Distribute Tasks Over Multiple Processor Cores

When at least two tasks are defined in your application, for example, `MainTask` (IEC task) and `LowTask` (IEC task) proceed as follows to distribute them over multiple processor cores.

Step	Action
1	Open the **Task Configuration** editor.
2	Open the **Task Groups** tab, page 405. **Result**: The default **IEC-Tasks** group is displayed.
3	Click the **Add Group** button. **Result**: A new task group with the default name `NewGroup` is created.
4	Double-click the `NewGroup` field and change the name to `LowGroup`.
5	From the **Core** field list, select an assignment, for example, **1**.
6	Open the task `LowTask` (IEC task) in the editor.
7	From the **Task group** list, select the new group `LowGroup`. **Result**: The `LowTask` task is processed by processor **Core** = **1** as it is a member of group `LowGroup`.

Displaying the CPU Load per Processor Core in the Trace

To display the CPU load, refer to the Menu Commands Online Help: **Project > Add Object > DeviceTrace...**.

Multicore - Tools

Overview

Multicore is a term associated with parallel processing and refers to the use of several cores that are simultaneously engaged in the execution of a program. Parallel processing divides computational-intensive operations into smaller parts and distributes these tasks among several cores. The goal is to speed up processing of computationally-intensive operations.

Characteristics of computationally-intensive applications:

- Large amount of data to process.
- Complex algorithms requiring heavy computations.

We can understand the control program for a system, that was previously calculated on a single processor core, as a complex algorithm. The division of this complex program flow into several subprograms (task partitioning) that can be executed as independently as possible enables the reduction of the cycle times. This results in a better reaction time of the system because the sampling rates can be increased.

Goals of task partitioning:

- Evenly dividing the effort among the available cores.
- Minimization of contention for shared resources.

Up to now, the processing of very large amounts of data (for example, image processing in real time) has often been outsourced to specialized control components because the computing capacity of controllers with one processor core has not been sufficient to handle both parts of the data processing in the required time window.

The ability to use multiple processor cores out of the controller program brings additional possibilities. The parallel processing of data, distributed over several processor cores, makes it possible to execute complex processing sequences (for example, for image processing in real time) in a required time window. In many cases, it is no longer necessary to implement parallel file processing outside the controller with special tools and programming languages. Parallel data processing becomes an integral part of a controller program.

Example

The new possibilities are illustrated with the example of industrial image processing, that is based on the evaluation of taken images, that then influences the sequence of further production steps.

In this example, the image processing consists of three steps:

1. Image capture due to an external event.
2. Image analysis.
3. Decision on rejection as scrap or forwarding to subsequent process steps.

With single core controllers, the image analysis step is outsourced to an external system that is triggered by an external event. The result of the image analysis is provided through a physical input of the controller.

With multi-core controllers, the following application design is feasible:

Image analysis is performed in a library block that is triggered by a local input variable. This reduces the processing time for image evaluation to a minimum based on the number of available processor cores. The calculations are thus dynamically distributed to an optimal number of processor cores. Subsequently, the results are summarized in order to be able to make a decision about whether the present product meets the requirements. In addition to the computational effort for image processing, there is also the effort for distributing the data and the effort for summarizing the results. Therefore, the image processing efficiency does not increase linearly with the number of processor cores. The different efforts are to be evaluated to lead to shorter processing times per product.

Measurement Tools

The following tools are available to measure and compare the runtime of applications and libraries on single and multicore systems.

- **Trace Manager** (*CmpTrageMgr* component)

  For further information, refer to the section Displaying the CPU Load with **DeviceTrace** Objects in the Project in the Menu Commands Online Help.
- **Profiler** add-on

  The **Profiler** add-on is provided in the CODESYS Store and can be installed with the Schneider Electric Software Installer.

Atomic Operations

Atomic operations are mainly used to implement locking mechanisms (locking) used in the synchronization between tasks when accessing shared data.

An operation acting on shared memory is atomic if it completes in a single step relative to other task. When an atomic store is performed on a shared variable, no other task can observe the modification half-complete. When an atomic load is performed on a shared variable, it reads the entire value as it appeared at a single moment in time.

Typical processor instructions for implementing atomic operations are:

- Atomic load and store of integral IEC data types as, for example, LINT, __XWORD, POINTER, INTERFACE, …
- Multicore operators
  - TEST_AND_SET, page 745
  - __COMPARE_AND_SWAP, page 743
  - __XADD, page 744

CPU Load Tab

Overview

The **CPU Load** tab of the **Task Configuration** is available in online mode for multicore controllers. The load of the individual processor cores is displayed in the trace editor. For further information, refer to Displaying the CPU Load with **DeviceTrace** Objects in the Project in the Menu Commands Online Help.

To add more variables, open the trace configuration by clicking the **Configuration** link on the right-hand side.

Watch List Editor

What's in This Chapter

Watch View / Watch List Editor 423
Creating a Watch List 423
Watch List in Online Mode 424

Watch View / Watch List Editor

Overview

A watch list is a user-defined set of project variables. They are displayed in the watch view for monitoring, page 424 the values in a table. Also, writing and forcing of the variables is possible within the watch view.

Open a watch view via the **Watch** command submenu (by default in the **View** menu). It provides an editor for creating watch lists, page 423.

By default, you can set up 4 individual watch lists in the watch views **Watch 1**, **Watch 2**, **Watch 3**, **Watch 4**. View **Watch all Forces** in online mode gets filled, page 424 automatically with all currently forced values of the active application.

Creating a Watch List

Procedure

Create a watch list by executing the following actions:

Step	Action	Comment
1	Execute the command **View > Watch > Watch<n>** to open a **Watch** view.	–
2	Double-click the field in the column **Expression** and enter the variable to be watched either manually or via the **Input Assistant**. Syntax: `<device name>.<application name>.<object name>.<variable name>`	**NOTE:** If you enter the name of a structured variable, then in online mode the particular structure components are entered in further lines automatically.
3	Subsequently define all variables which you want monitored in the current watch list. You can modify the order of the entries via drag and drop.	The fields **Execution point**, **Type**, **Address**, **Comment** are filled automatically corresponding to the declaration of the variables. The symbol preceding the expression indicates whether it is an input variable , an output variable , or a normal variable .

Online Mode

In online mode, the current variable value is shown in the column **Value**. Like at other monitoring positions, you can define **Prepared Values** and force or write them. For further information, refer to the chapters *Force Values, Prepare Value Dialog Box* (see Menu Commands, Online Help) and *Monitoring*, page 251.

In online mode, you can add a currently focused expression to the watch list by executing the command **Add Watch**.

Watch List in Online Mode

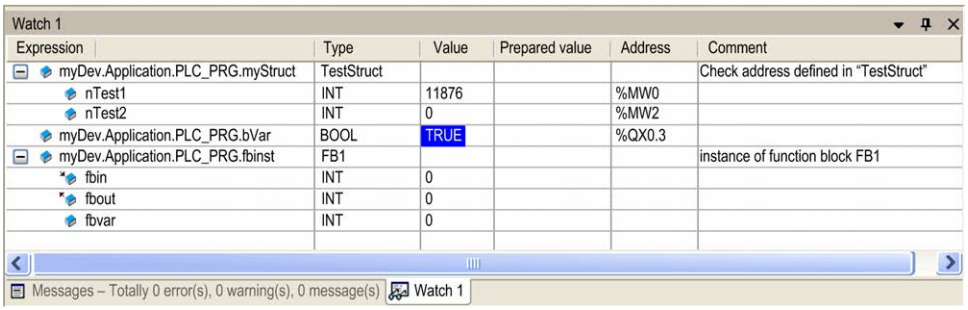
Monitoring

A watch list, page 423 (**Watch<n>**) in online mode displays the current value of a variable in the **Value** column. This is the value the variable has between two task cycles.

Also any assigned direct IEC address and/or comment are displayed. The components of the view correspond to those of the online view of the declaration editor, page 378.

See chapter *Creating a Watch List*, page 423 for a description on how to set up such a watch list and how to handle folds in case of structured variables.

Watch view in online mode



NOTE: In online mode you can add expressions to the watch list by use of the command **Add Watch**.

Write and Force Values

In column **Prepared value**, you can enter a desired value which will be written or forced to the respective expression on the controller by command **Write values** or **Force values**. Refer to the descriptions of the commands **Write** and **Force**, usable also in other monitoring views (for example, declaration editor).

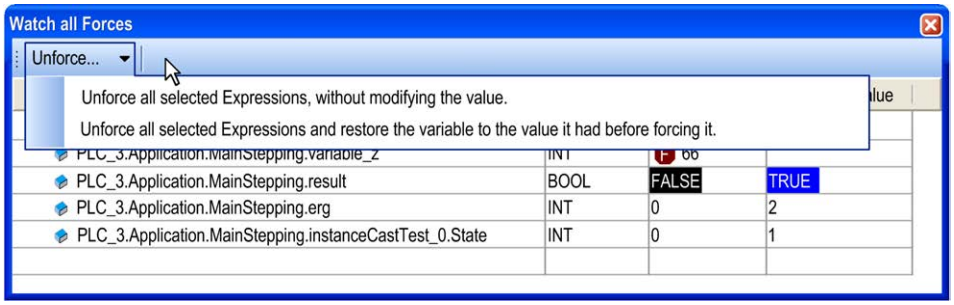
Watch All Forces

This is a special watch list view, which in online mode is automatically filled with all currently forced values of the active application. Each **Expression**, **Type**, **Value**, and **Prepared value** will be shown, as in the online view of a **Watch<n>** list.

You can unforce values by one of the following commands available via the button **Unforce...**:

- **Unforce all selected Expressions, without modifying the value.**
- **Unforce all selected Expressions and restore the variable to the value it had before forcing it.**

Watch all Forces dialog box



Tools Within Logic Editors

What's in This Chapter

Function and Function Block Finder 426
Input Assistant 429

Function and Function Block Finder

Overview

EcoStruxure Machine Expert provides the FFB (function and function block) finder that assists you in finding a specific function or function block even if you do not know its exact name.

You can use the function and function block finder in the following programming languages that allow to insert function blocks:

- CFC
- LD
- IL
- FBD
- ST

How to Find a Function or Function Block with the FFB Finder

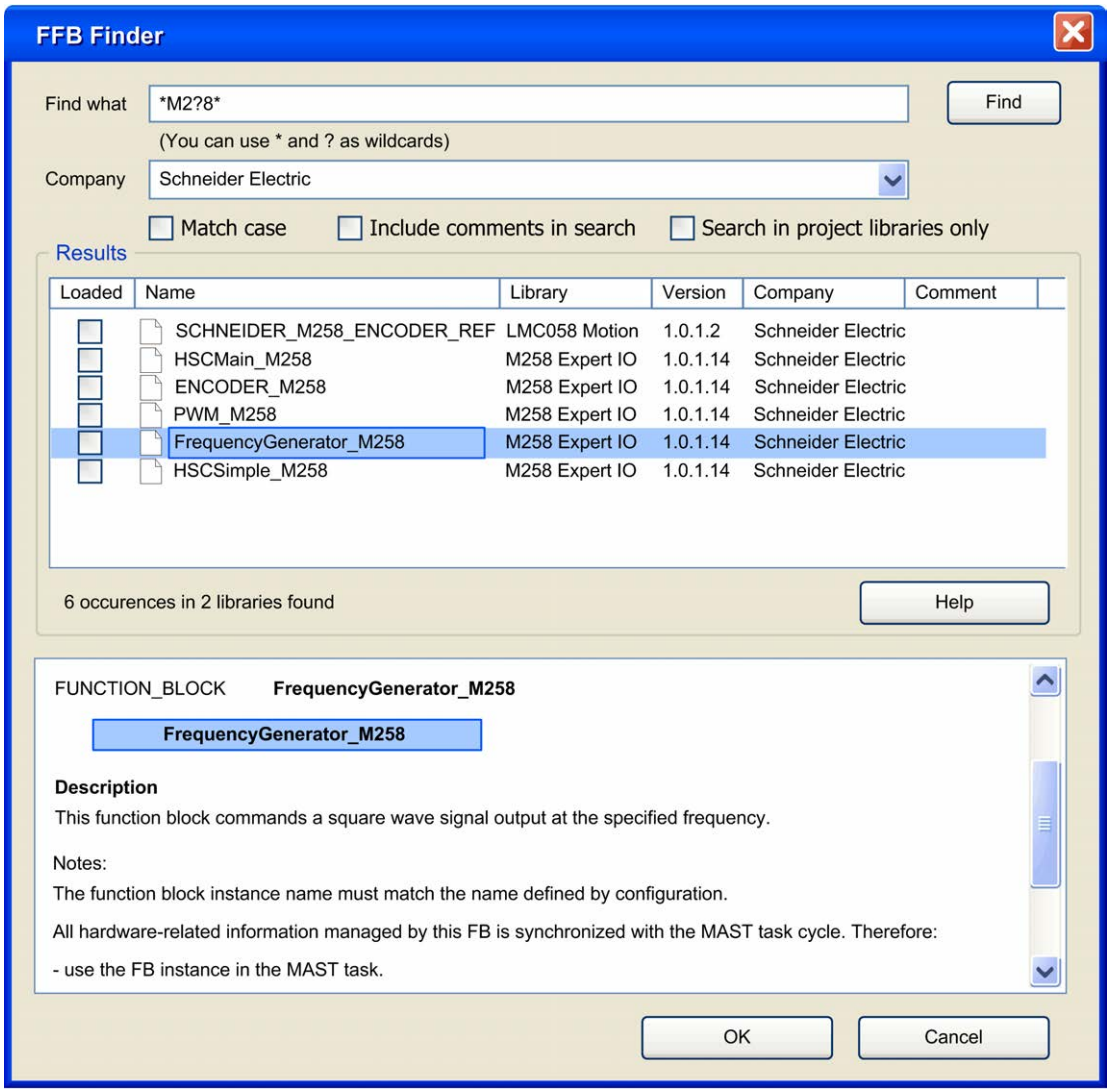
When you are about to create programming code in the Logic Builder, go to the place where you want to insert the function block and open the FFB finder as follows:

- select the menu **Edit > FFB Finder**

  or

- right-click at the respective place in the editor and select the command **FFB Finder...** from the contextual menu

The **FFB Finder** dialog box opens:



The **FFB Finder** dialog box contains the following elements for finding a function or function block:

Element	Description
Find what	In the **Find what** textbox, enter the name of the function or function block you want to insert into your programming code. As wildcards you can use a question mark (?), which replaces exactly one character, or an asterisk (*), which can replace several characters or no character at all.
Company	If you know the company that created the library which includes the function block you are searching for, you can select the companies from the **Company** list. This parameter is by default set to **All companies**.
Match case	Check the **Match case** check box to perform a case-sensitive search. By default, this check box is not selected.
Include comments in search	Check the **Include comments in search** check box to search for the entered string not only in the names of functions and function blocks but also in the comments that are saved with them. By default, this check box is not selected.
Search in project Libraries only	Check the **Search in project libraries only** check box to limit the search to those libraries that are used in the current application. By default, this check box is not selected and the find operation includes all libraries that are installed on the EcoStruxure Machine Expert PC.
Find	Click the **Find** button or press the ENTER key to start searching for the function or function block.

Results Returned by the FFB Finder

Any function or function block that matches the entered search criteria will be listed in the **Results** list with the following information:

- **Name** of the function or function block
- the **Library** the function or function block is saved in
- the **Version** of the library
- the **Company** that created the library
- A comment, if available, will be displayed in the column on the right side.
- The column **Loaded** on the left side indicates whether the library, the function or function block is saved in, is already used in the current project.

To display further information on one of the functions or function blocks, select it from the list. In the field below, a graphic of the function / function block with its inputs and outputs will be displayed, as well as a description or any further information, if available.

Integrating a Function / Function Block into the Programming Code

To integrate a function / function block that was found by the FFB Finder in your programming code, select it in the **Results** list, and

- either double-click the selected entry in the **Results** list,
- or click the **OK** button.

The selected function / function block will be inserted at the place where your cursor is positioned in your programming code and the respective library will be loaded automatically.

Repeat this operation whenever you need assistance in finding a specific function / function block.

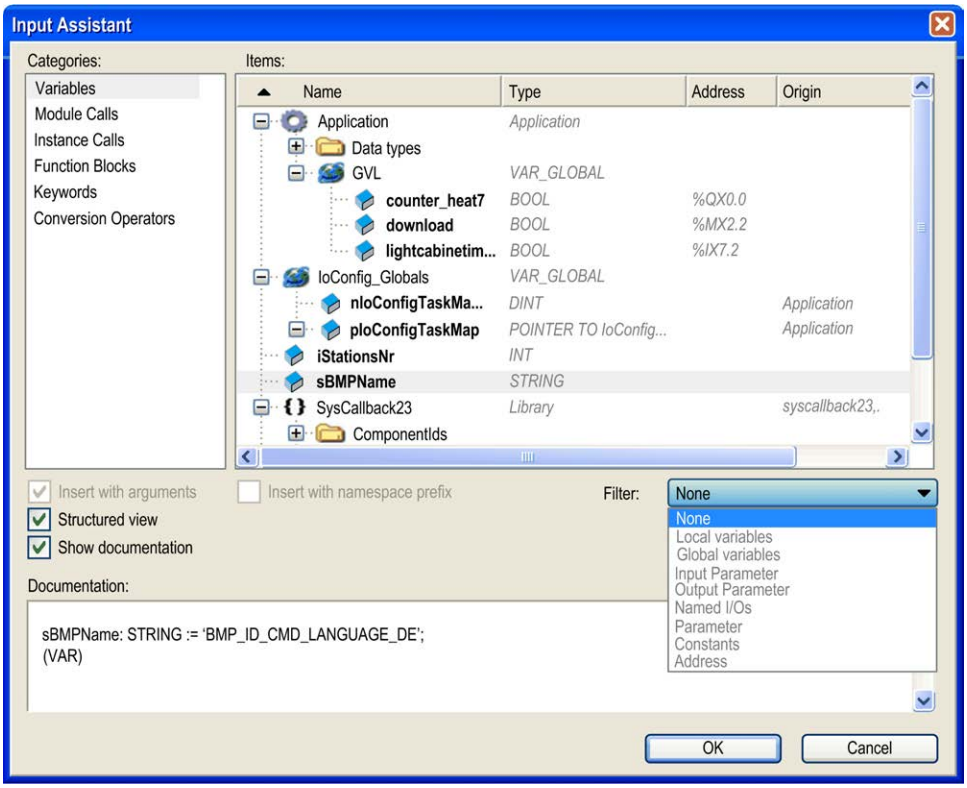
Input Assistant

Overview

The **Input Assistant** dialog box and the corresponding command **Input Assistant** (by default in the **Edit > Smart Coding** menu) are only available if the cursor is placed in a text editor window. The dialog box offers the available project items for being inserted at the current cursor position.

Default shortcut: **F2**

Input Assistant dialog box



Description of the Elements

The **Input Assistant** dialog box provides the following elements:

Element	Description
Categories	In this area, the project items are sorted by **Categories**.
Filter	You can set a **Filter** for the category **Variables**. To display a certain type of variable, select an entry from the list, such as **Local variables**, **Global variables**, **Constants**.
Items area	
Name, **Type**, **Address**, **Origin**	The **Items** area shows the available items and - depending on the category - also their data **Type**, **Address**, and **Origin** for the category selected in the **Categories** area. The **Origin** is shown for I/O variables (path within the **Devices Tree**) and library-defined variables (library name and category). You can sort the items by **Name**, **Type**, **Address**, **Origin** in ascending or descending alphabetic order. To achieve this, click in the respective column header (arrow-up or arrow-down symbol). To hide or display the columns **Type**, **Address** or **Origin**, right-click the headline of the respective column.
Structured view	If the option **Structured view** is selected, the project items are displayed in a structure tree supplemented with icons. If the option is not selected, the project items are arranged flat. Each project item is displayed with the POU it belongs to (example: **GVL1.gvar1**).

NOTE: If there are objects with the same name available in the **Global** node of the **Applications Tree** as well as below an application (**Applications Tree**), only 1 entry is offered in the **Input Assistant** because the usage of the object is determined by the usual call priorities (first the application-assigned object, then the global one).

Element	Description
Show documentation	If the option **Show documentation** is selected, the **Input Assistant** dialog box is extended by the **Documentation** field. If the selected element is a variable and an address is assigned to this variable or a comment has been added at its declaration, these are displayed here.
Insert with arguments	If this option is selected, items which include arguments, for example functions, are inserted with those arguments. Example: If function block *FB1*, which contains an input variable *fb1_in* and an output variable *fb1_out*, is inserted with arguments, the following will be written to the editor: `fb1(fb1_in:= , fb1_out=> )`
Insert with namespace prefix	If this option is selected, the item is inserted with the prefixed namespace. This option has to be selected when inserting objects into libraries which have defined in their **Properties** that the use of the namespace prefix is obligatory (option **Only allow qualified access to all identifiers** (see Functions and Libraries User Guide) is selected).

Tools

What’s in This Part

Data Logging 432
Recipe Manager 434
Trace Editor 454
Trend Recording 472
Alarm Configuration 481
Unit Conversion 501
Symbol Configuration Editor 505
Data Exchange: Controller - HMIs, OPC DA and UA Server 517
Cam Motion Editor 529
Script Hook Manager 546

Data Logging

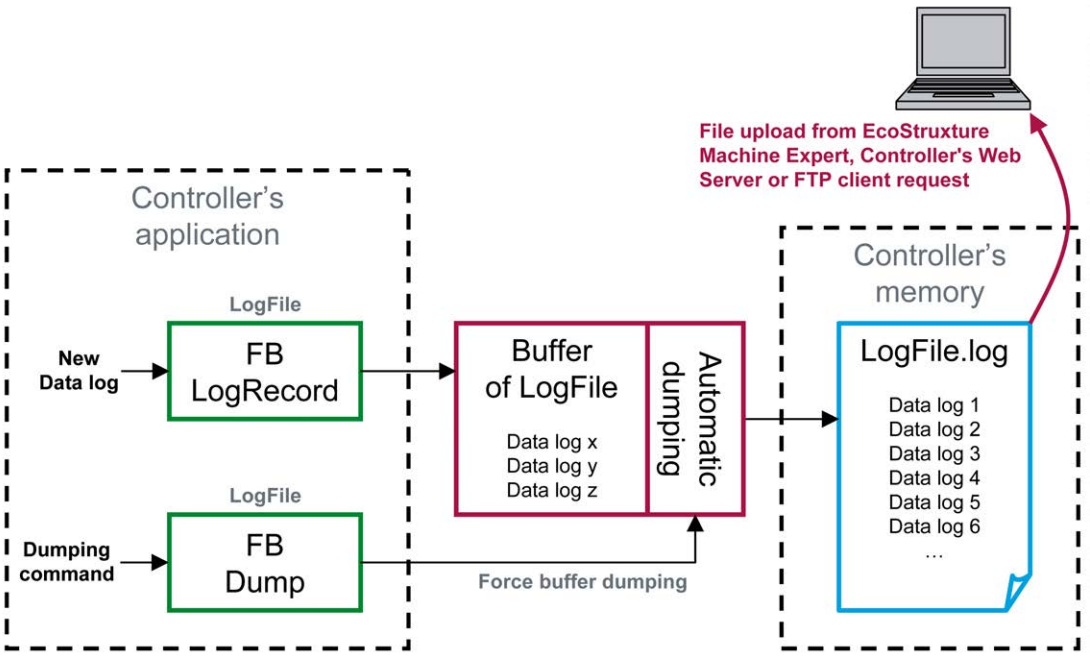
What's in This Chapter

Introduction to Data Logging 432

Introduction to Data Logging

Overview

You can monitor and analyze application data by examining the data log file (.log).



The figure shows an application that includes the 2 function blocks, `LogRecord` and `Dump`. The `LogRecord` function block writes data to the buffer, which empties into the data log file (.log) located into the controller memory. The buffer dumping is automatic when 80% full or it can be forced by the `Dump` function. As a standard FTP client, a PC can access this data log file when the controller acts as an FTP server. It is also possible to upload the file with EcoStruxure Machine Expert.

NOTE: Only controllers with file management functionality can support data logging. Refer to your controller programming manual to see if it supports file management.

Sample Data Log File (.log)

```
Entries in File: 8; Last Entry: 8;
18/06/2009;14:12:33;cycle: 1182;
18/06/2009;14:12:35;cycle: 1292;
18/06/2009;14:12:38;cycle: 1450;
18/06/2009;14:12:40;cycle: 1514;
18/06/2009;14:12:41;cycle: 1585;
18/06/2009;14:12:43;cycle: 1656;
18/06/2009;14:14:20;cycle: 6346;
18/06/2009;14:14:26;cycle: 6636;
```

Implementation Procedure

First declare and configure the data log files in your application before starting to write your program.

Recipe Manager

What's in This Chapter

Recipe Manager 435
Recipe Definition 439
RecipeManCommands 444
Loading Recipe Values from the Controller 452
Memory Usage by Recipes 453

Recipe Manager

Overview

The **Recipe Manager** provides the functionality for handling user-defined lists of project variables, named recipe definitions, and definite value sets for these variables within a recipe definition, named recipes.

You can use recipes to modify or read recipe values for a specific set of variables (recipe definition) on the controller. They can also be loaded from and saved to files. These interactions are possible by using visualization elements which you have to configure appropriately (input configuration execute command). You can also use certain recipe commands in the application, page 444.

When you have selected a recipe, validate that the recipe is appropriate for the process that will be controlled.

⚠WARNING

UNINTENDED EQUIPMENT OPERATION

- Conduct a safety analysis for the application and equipment installed.
- Verify that recipe is appropriate for the process and equipment or function in the installation.
- Supply appropriate parameters, particularly for limits and other safety-related elements.
- Verify that all sensors and actuators are compatible with the recipe selected.
- Thoroughly test all functions during verification and commissioning.
- Provide independent paths for critical control functions (emergency stop, over-limit conditions etc.) according to the safety analysis and applicable codes and regulations.

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

By default, the recipe manager is loaded to the controller during download. It handles the writing and reading of recipes when the application is running on the controller. However, it is not necessary to load the recipe manager to the controller to use recipes only for exchanging parameters during startup of the system (that is when EcoStruxure Machine Expert is still connected to the controller). You can deactivate its download for this purpose using the option **Recipe Management in the PLC**. The writing and reading of recipe values will then be handled by the standard online commands and services. If the recipe management has to run on the controller, because it is needed by the application program during run time, then the *RecipeCommands* function block is responsible for handling the recipe commands.

For a description of the behavior of recipes in the various online modes, refer to the chapter *Recipe Definition*, page 439.

If the recipe manager is located on another controller other than the application being affected by the recipes, the data server will be used to read/write the variables contained in the recipes. Reading and writing of the variables is done synchronously. By calling *g_RecipeManager.LastError* after reading/writing, you may verify if the transmission has been carried out successfully (*g_RecipeManager.LastError=0* in this case).

Recipe Management Objects in the Tools Tree

To add a **Recipe Manager** object to the **Tools tree**, select the **Application** node, click the green plus button, and execute the command **Add other objects... > Recipe Manager...**. Confirm the **Add Recipe Manager** dialog box by clicking **Add** and the **Recipe Manager** node is inserted below the **Application** node.

You can add one or several **Recipe Definition** objects to a **Recipe Manager** node. To achieve this, click the green plus button of the **Recipe Manager** node and execute the command **Recipe Definition...**. Enter a **Name** in the **Add Recipe Definition** dialog box, and click **Add**. Double-click the node to view and edit recipe definitions including the particular recipes in a separate editor window. For a description of the behavior of recipes in the various online modes, refer to the chapter *Recipe Definition*, page 439.

Recipe Manager Editor, Storage Tab

By default, the recipes will be stored automatically to files according to the settings in the **Storage** tab of the **Recipe Manager** editor.

Parameter	Description
Storage Type	Select **Textual** or **Binary** storage type.
File Path	Specify the location where the recipe is to be saved in the controller or in the local file system. **NOTE:** For recipe managers that work in simulations, the file path must start with a dot (.), for example, `./Rcp`.
File Extension	Specify the file extension of the recipe file.

NOTE: A storage file can also be defined by the input on a visualization element (input configuration - execute command - save/load a recipe from a file). However, when defining the name of such a file in the visualization configuration, do not overwrite the * *.txtrecipe* / * *.rcp* file defined here in the recipe manager.

Parameter	Description
Separator	In case of textual storage, the columns selected for storage will be separated by a separator. Select 1 of the 6 options proposed.
Available Columns	All columns of the recipe definition, represented by the respective header.
Selected Columns	Selected columns of the recipe definition, that is, the columns to be stored. At least the column containing the **Current Value** is included in this part. It cannot be deselected.
arrow buttons	The other columns can be shifted to the right or to the left by selecting the respective entry and clicking the arrow buttons. You can also shift all entries from one side to the other at once by using the double arrow buttons.
Up and **Down** buttons	Click these buttons to adjust the order of the selected columns, which represents the order of the columns in the storage file. For each recipe, a file <recipe name>.<recipe definition>.<file extension> will be created in the specified folder. This file will be reloaded to the recipe manager at each restart of the application. For the update configuration of the recipe storage files, refer to the description of the **General** tab, page 438.
Save as default	Click the **Save as default** button to use the settings made within this dialog box as default settings for each further recipe manager inserted.

NOTE: Floating point values (type REAL/LREAL) are stored in the textual recipe files in decimal format as well as in hexadecimal format. (Because the hexadecimal value represents the exact value whereas the decimal REAL value represents the value to the seventh decimal place.)

Example: `PLC_PRG.realVar:=22.0F16#1600000H-5`

For manually modifying a value in the recipe file, edit the decimal value and remove the subsequent hexadecimal entry. (If both values are available, the hexadecimal value is loaded.)

Recipe Manager Editor, General Tab

Parameter	Description
Recipe Management in the PLC	If the recipe manager is not needed on the controller because no recipes are to be handled during run time of the application, you can deactivate this option so that the manager is not downloaded. An automatic update of the recipe file is only possible after the download has been performed. To download the recipe management to the controller, select this option.
Save recipe	
Save recipe changes to recipe files automatically	This option is available only if the option **Recipe Management in the PLC** is selected. Select the **Save recipe changes to recipe files automatically** option to update recipe files automatically in runtime mode whenever a recipe is changed.
Load recipe	If the option **Recipe Management in the PLC** is activated, the following two options are available to configure the download from the controller.
Load only by exact match of the variable list	Select the option to load recipe files only if the file contains all variables available in the variable list of the recipe definition of the application. The variables in the file must be in the same order as in the variable list. Additional entries at the end are ignored. Otherwise, the recipe cannot be loaded and the return value, page 446 *ERR_RECIPE_MISMATCH* is set (*RecipeManCommands.GetLastError*).
Load matching variables by variable name	Select the option to load only variables with matching variable names from the recipe file. No error status is set when the variable list in the file is different from the list in the recipe definition of the application. Thus, recipe files can also be loaded if variables in the file or in the recipe definition have been deleted.
Overwrite existing recipes on download	Select the option to overwrite recipe files available on the controller with the values configured in the project when the application is started. If the option is not selected, the values of the existing recipe files are loaded. As a prerequisite, the parameter **Storage Type** is set to **Textual** and the option **Save recipe changes to recipe files automatically** is selected.
Write recipe	
Limit the variable to min/max when recipe value is out of the range	Select the option to write the defined minimum or maximum value to the controller if the recipe contains a value that is beyond the value range.
Do not write to a variable when the recipe value is out of the min/max range	Select the option to prevent a value from being written to the controller if the recipe contains a value that is beyond the value range. The value that is present in the controller will be retained.
Read recipe	
Check recipe for changes	Use the function block *RecipeManCommands* of the RecipeManagement library to read recipes. Do not call the method cyclically as this is time and controller capacity-consuming. Select the option to read the variable values present in the controller into the recipe with each method call. Only if the values have changed, the recipe file is overwritten with the present values. As this procedure generates additional code for comparing the old and the present values, consider that it has an impact on the performance. If the option is not selected, the variable values present in the controller are read into the recipe with each method call. The recipe is then written to the recipe file in the local file system. **NOTE:** This write process to the local file system with each method call may have a considerable impact on the controller performance.

Recipe Definition

Overview

The Recipe Manager, page 435 handles 1 or several recipe definitions. A recipe definition contains a list of variables and 1 or several recipes (value sets) for these variables. You can save a recipe to a file or write recipe files to the controller. By using different recipes, you can assign another set of values to a set of variables on the controller in one stroke. There is no limitation of the number of recipe definitions, recipes, and variables per recipe.

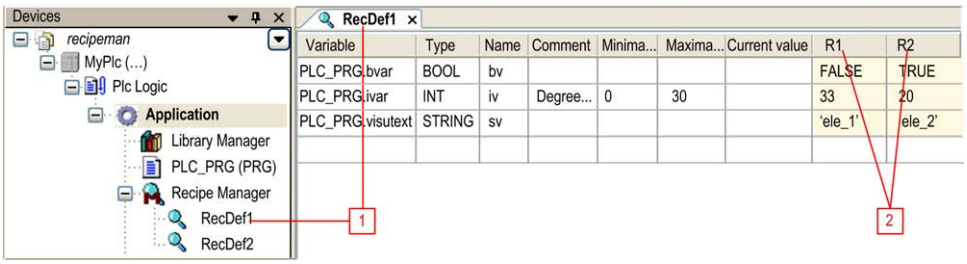
NOTE: The recipe management reads the values of the variables that are defined in the recipe definition at the end of the initialization phase of the application. Since the initial values of the application variables have all been set at this point, missing values from recipe files can be correctly initialized.

Recipe Definition

You can add one or several **Recipe Definition** objects to a **Recipe Manager** node in the **Tools tree**. To achieve this, click the green plus button of the **Recipe Manager** node and execute the command **Recipe Definition...**.

Double-click the node to view and edit recipe definitions including the particular recipes in a separate editor view.

Recipe definition editor view



1 Recipe definition name

2 Recipe names

The editor window is titled with the name of the recipe definition.

Parameter	Description
Variable	In the table, you can enter several project variables for which you want to define 1 or several recipes. For this purpose, you can use the command **Insert Variable** when the cursor is in any field of any line. Alternatively, you can double-click a **Variable** field, or you can select it and press the spacebar to get into editor mode. Enter the valid name of a project variable, for example *plc_prg.ivar*. You can click the **...** button to open the **Input Assistant**. You can also specify a POU, for example, a program like `PLC_PRG` shown in the figure above. In this case, all variables defined within the POU are added to the recipe definition automatically when you close the input field. The same applies to data types or function blocks. You can toggle between the normal and the structured view using the buttons on the right side. After you have modified the declaration of structured data types or POUs, the recipe definition can be reduced or extended by the concerned variables automatically. For further information, refer to the **Update structured variables** command (see Menu Commands, Online Help).
Type	The **Type** field is filled automatically. Optionally, you can define a symbolic **Name**.
Name	You can define a symbolic **Name**.
Comment	Enter additional information, such as the unit of the value recorded in the variable.
Minimal Value and **Maximal Value**	You can optionally specify these values which should be permissible for being written on this variable.
Current Value	This value is monitored in online mode.
Save recipe changes to recipe files automatically	It is a good practice to activate this option in the **General** tab of the **Recipe manager** editor because it affects the usual behavior of a recipe management: the storage files are updated immediately at any modification of a recipe during run time. Consider that the option can only be effective as long as the recipe manager is available on the controller.

You can remove a variable (line) from the table by pressing the **Delete** key when one of its cells is selected. You can select multiple lines by keeping the **Ctrl** key pressed while selecting cells. You can copy the selected lines by copy and paste. The paste command inserts the copied lines above the selected line. In doing so, recipe values are inserted in the matching recipe column, if available.

To add a recipe to the recipe definition, execute the **Add a new recipe** command (see Menu Commands, Online Help) when the focus is in the editor view. For each recipe, an own column is created, titled with the recipe name (example: **R1** and **R2** in the figure above).

In online mode, a recipe can be changed either by an appropriately configured visualization element (input configuration execute command) or by using the appropriate methods of the function block *RecipeManCommands* of the Recipe_ Management.library.

For a list of methods available in the contextual menu of a recipe column in the recipe definition editor view, refer to Using Recipes in Online Mode, page 442.

See in the following paragraphs how the recipes behave in the particular online states. It is a good practice to set the option **Save recipe changes to recipe files automatically** (in order to get the usual behavior of a recipe management).

Recipe

You can add or remove a recipe offline or online. In offline mode, use the commands **Add a new recipe** (see Menu Commands, Online Help) and **Remove recipes** (see Menu Commands, Online Help) within the recipe manager editor. In online mode, either configure an input on an appropriately configured visualization element, or use the appropriate methods of function block *RecipeManCommands* of the Recipe_Management.library.

When adding a recipe, a further column is added behind the right-most column, titled with the name of the recipe (see the figure of the recipe definition editor view). The fields of a recipe column can be filled with appropriate values. Thus, for the same set of variables, different sets of values can be prepared in the particular recipes.

Using Recipes in Online Mode

The recipes can be handled (created, read, written, saved, loaded, deleted) by using the methods of the function block *RecipeManCommands*, provided by the library Recipe_Management.libray, in the application code, or via inputs on visualization elements.

Recipe handling in online mode if **Save recipe changes to recipe files automatically** is activated:

Actions	**Recipes Defined Within the Project**	**Recipes Created During Run Time**
Online Reset Warm **Online Reset Cold** **Download**	The recipes of all recipe definitions get set with the values out of the open project.	Dynamically created recipes remain unchanged.
Online Reset Origin	The application is removed from the controller. If a new download is done afterwards, the recipes will be restored like on an **Online Reset Warm**.	
Shut down and restart the controller	After the restart, the recipes are reloaded from the automatically created files. So the status before shutdown will be restored.	
Online Change	The recipe values remain unchanged. During run time, a recipe can only be modified by the commands of the *RecipeManCommands* function block.	
Stop	At a stop/start of the controller, the recipes remain unchanged.	

NOTE: Floating point values (type REAL/LREAL) are stored in the textual recipe files in decimal format as well as in hexadecimal format. (Because the hexadecimal value represents the exact value whereas the decimal REAL value represents the value to the seventh decimal place.)

Example: `PLC_PRG.realVar:=22.0F16#1600000H-5`

For manually modifying a value in the recipe file, edit the decimal value and remove the subsequent hexadecimal entry. (If both values are available, the hexadecimal value is loaded.)

Recipe handling in online mode if **Save recipe changes to recipe files automatically** is NOT activated:

Actions	Recipes Defined Within the Project	Recipes Created During Run Time
Online Reset Warm **Online Reset Cold** **Download**	The recipes of all recipe definitions get set with the values out of the open project. However, these are only set in the memory. In order to store the recipe in a file, the **Save Recipe** command must be used explicitly.	Dynamically created recipes get lost.
Online Reset Origin	The application is removed from the controller. If a new download is done afterwards, the recipes will be restored.	Dynamically created recipes get lost.
Shut down and restart the controller	After the restart the recipes are reloaded from the initial values which had been created at download from the values out of the project. So the status as it was before shutdown will not be restored.	
Online Change	The recipe values remain unchanged. During run time, a recipe can only be modified by the commands of the *RecipeManCommands* function block.	
Stop	At a stop/start of the controller, the recipes remain unchanged.	

Further information:

- Concerning the storage of recipes in files, which are reloaded at a restart of the application, refer to the description of the *Recipe Manager Editor, Storage Tab*, page 436.
- For a description of the particular *RecipeManCommands* methods, page 444, refer to the documentation within the library.
- For the input configuration of a visualization element, refer to the help page (category **Input > execute command**).

The following actions on recipes are possible:

Actions	Description
Create recipe (= **Add a new recipe**)	A new recipe is created in the specified recipe definition.
Read recipe	The values of the variables of the specified recipe definition are read from the controller and are written to the specified recipe. Therefore, the values will be stored implicitly (in a file on the controller). They will also be monitored immediately in the recipe definition table in the **Recipe Manager**. In other words, the recipe managed in the **Recipe Manager** gets updated with the actual values from the controller.
Write recipe	The values of the given recipe, as visible in the recipe manager, are written to the variables on the controller.
Save Recipe	The values of the specified recipe are written to a file with extension **.txtrecipe* or **.rcp*, the name of which you have to define. For this purpose, the dialog box for saving a file in the local file system opens. **NOTE:** The implicitly used recipe files, necessary as a buffer for reading and writing of the recipe values, may not get overwritten. Therefore, the name for the new recipe file must be different from <recipe name>.<recipe definition name>.txtrecipe / .rcp.
Load Recipe	The recipe which has been stored in a file (see the **Save Recipe** description) can be reloaded from this file. The dialog box for browsing for a file opens for this purpose. The filter is automatically set to extension **.txtrecipe* / **.rcp*. After reloading the file, the recipe values will be updated accordingly in the recipe manager. For further information, refer to the description of the **Load Recipe** command in the Menu Commands Online Help (see Menu Commands, Online Help).

Actions	Description
Delete recipe (= **Remove recipe**)	The specified recipe is removed from the recipe definition.
Change recipe	The value of the project variables can be changed. With a following write recipe action, the project variables are written with the new values.

NOTE:

When using recipe files (create, read, write, delete), create specific tasks, page 413 with low priority and with the **Watchdog** function disabled.

RecipeManCommands

Overview

When calling a recipe command, internal data access is performed. Depending on the device type, this takes a few milliseconds. Verify that these calls are not performed by the MAST task or by a task with a configured watchdog or a real-time task. This may lead to an application error and the controller will enter the HALT state.

Consider that the option **Save recipe changes to recipe files automatically** also performs a file access with each modification of the recipe. Deactivate this option if the storage of the recipe is triggered by the application.

Return Values

For possible return values for recipe commands, refer to the *Recipe Management Library Documentation* in the **CODESYS Libraries** part of the online help.

CreateRecipe

This method creates a new recipe in the specified recipe definition and afterwards reads the controller values into the new recipe. At the end, the new recipe is stored in the default file.

Parameter	Data type	Description
RecipeDefinitionName:	STRING	Name of the recipe definition.
RecipeName:	STRING	Name of the recipe.

Return values, page 444: *ERR_NO_RECIPE_MANAGER_SET, ERR_RECIPE_DEFINITION_NOT_FOUND, ERR_RECIPE_ALREADY_EXIST, ERR_FAILED, ERR_OK*

CreateRecipeNoSave

This method creates a new recipe in the specified recipe definition and afterwards reads the controller values into the new recipe.

Parameter	Data type	Description
RecipeDefinitionName:	STRING	Name of the recipe definition.
RecipeName:	STRING	Name of the recipe.

Return values, page 444: *ERR_NO_RECIPE_MANAGER_SET, ERR_RECIPE_DEFINITION_NOT_FOUND, ERR_RECIPE_NOT_FOUND, ERR_FAILED, ERR_OK*

DeleteRecipe

This method deletes a recipe from a recipe definition.

Parameter	Data type	Description
RecipeDefinitionName:	STRING	Name of the recipe definition.
RecipeName:	STRING	Name of the recipe.

Return values, page 444: *ERR_NO_RECIPE_MANAGER_SET, ERR_RECIPE_DEFINITION_NOT_FOUND, ERR_RECIPE_NOT_FOUND, ERR_FAILED, ERR_OK*

DeleteRecipeFile

This method deletes the default recipe file from a recipe.

Parameter	Data type	Description
RecipeDefinitionName:	STRING	Name of the recipe definition.
RecipeName:	STRING	Name of the recipe.

Return values, page 444: *ERR_NO_RECIPE_MANAGER_SET, ERR_RECIPE_DEFINITION_NOT_FOUND, ERR_RECIPE_NOT_FOUND, ERR_RECIPE_FILE_NOT_FOUND, ERR_OK*

LoadAndWriteRecipe

This method loads a recipe from the default recipe file and afterwards writes the recipe into the controller variables.

Parameter	Data type	Description
RecipeDefinitionName:	STRING	Name of the recipe definition.
RecipeName:	STRING	Name of the recipe.

Return values, page 444: *ERR_NO_RECIPE_MANAGER_SET, ERR_RECIPE_DEFINITION_NOT_FOUND, ERR_RECIPE_NOT_FOUND, ERR_RECIPE_FILE_NOT_FOUND, ERR_RECIPE_MISMATCH, ERR_FAILED, ERR_OK*

LoadFromAndWriteRecipe

This method loads a recipe from the specified recipe file and afterwards writes the recipe into the controller variables.

Parameter	Data type	Description
RecipeDefinitionName:	STRING	Name of the recipe definition.
RecipeName:	STRING	Name of the recipe.
FileName:	STRING [255]	Name of the file.

Return values, page 444: *ERR_NO_RECIPE_MANAGER_SET, ERR_RECIPE_DEFINITION_NOT_FOUND, ERR_RECIPE_NOT_FOUND, ERR_RECIPE_FILE_NOT_FOUND, ERR_RECIPE_MISMATCH, ERR_FAILED, ERR_OK*

LoadRecipe

This method loads a recipe from the default recipe file. The default recipe file name is <recipe>.<recipe definition>.<recipeextension>.

Parameter	Data type	Description
RecipeDefinitionName:	STRING	Name of the recipe definition.
RecipeName:	STRING	Name of the recipe.

Return values, page 444: *ERR_NO_RECIPE_MANAGER_SET, ERR_RECIPE_DEFINITION_NOT_FOUND, ERR_RECIPE_NOT_FOUND, ERR_RECIPE_FILE_NOT_FOUND, ERR_RECIPE_MISMATCH, ERR_FAILED, ERR_OK*

ReadAndSaveAs

This method reads the controller values from the variables of the recipe definitions and afterwards stores the data set to a file recipe without modifying the default recipe file <recipe.recipe definition.extension>.

Parameter	Data type	Description
RecipeDefinitionName:	STRING	Name of the recipe definition. The variables defined in the recipe definition are read.
FileName:	STRING[255]	Name of the file in which the read data set is saved as a recipe.

Return values, page 444: *ERR_NO_RECIPE_MANAGER_SET, ERR_RECIPE_DEFINITION_NOT_FOUND, ERR_RECIPE_SAVE_ERR, ERR_OK*

ReadAndSaveRecipe

This method reads the controller values into the recipe and afterwards stores the recipe into the default recipe file.

Parameter	Data type	Description
RecipeDefinitionName:	STRING	Name of the recipe definition.
RecipeName:	STRING	Name of the recipe.

Return values, page 444: *ERR_NO_RECIPE_MANAGER_SET, ERR_RECIPE_DEFINITION_NOT_FOUND, ERR_RECIPE_NOT_FOUND, ERR_RECIPE_SAVE_ERR, ERR_FAILED, ERR_OK*

ReadAndSaveRecipeAs

This method reads the controller values into the recipe and afterwards stores the recipe into the specified recipe file. The content of an existing file would be overridden.

Parameter	Data type	Description
RecipeDefinitionName:	STRING	Name of the recipe definition.
RecipeName:	STRING	Name of the recipe.
FileName:	STRING [255]	Name of the file.

Return values, page 444: *ERR_NO_RECIPE_MANAGER_SET, ERR_RECIPE_DEFINITION_NOT_FOUND, ERR_RECIPE_NOT_FOUND, ERR_RECIPE_SAVE_ERR, ERR_FAILED, ERR_OK*

SaveRecipe

This method stores the recipe into the default recipe file. The content of an existing file would be overridden. The default recipe file name is <recipe>.<recipedefinition>.<recipeextension>.

Parameter	Data type	Description
RecipeDefinitionName:	STRING	Name of the recipe definition.
RecipeName:	STRING	Name of the recipe.

Return values, page 444: *ERR_NO_RECIPE_MANAGER_SET, ERR_RECIPE_DEFINITION_NOT_FOUND, ERR_RECIPE_NOT_FOUND, ERR_RECIPE_SAVE_ERR, ERR_FAILED, ERR_OK*

ReadRecipe

This method reads the controller values into the recipe.

Parameter	Data type	Description
RecipeDefinitionName:	STRING	Name of the recipe definition.
RecipeName:	STRING	Name of the recipe.

Return values, page 444: *ERR_NO_RECIPE_MANAGER_SET, ERR_RECIPE_DEFINITION_NOT_FOUND, ERR_RECIPE_NOT_FOUND, ERR_FAILED, ERR_OK*

WriteRecipe

This method writes the recipe into the controller variables.

Parameter	Data type	Description
RecipeDefinitionName:	STRING	Name of the recipe definition.
RecipeName:	STRING	Name of the recipe.

Return values, page 444: *ERR_NO_RECIPE_MANAGER_SET, ERR_RECIPE_DEFINITION_NOT_FOUND, ERR_RECIPE_NOT_FOUND, ERR_FAILED, ERR_OK*

ReloadRecipes

This method reloads the list of recipes from the file system. Since the **Recipe Manager** does not synchronize with the file system automatically, use this method programmatically to notify the **Recipe Manager** regarding the change of recipe definition.

Parameter	Data type	Description
RecipeDefinitionName:	STRING	Name of the recipe definition.

Return values, page 444: *ERR_NO_RECIPE_MANAGER_SET, ERR_RECIPE_DEFINITION_NOT_FOUND, ERR_FAILED, ERR_OK*

GetRecipeCount

This method returns the number of recipes from the corresponding recipe definition.

Parameter	Data type	Description
RecipeDefinitionName:	STRING	Name of the recipe definition.

GetRecipeNames

This method returns the recipe names from the corresponding recipe definition.

Parameter	Data type	Description
RecipeDefinitionName:	STRING	Name of the recipe definition.
pStrings :	POINTER TO ARRAY [] OF STRING	The strings where the recipe values should be stored.
iSize :	INT	The size of an array of strings.
iStartIndex :	INT	The start index; can be used for a scrolling function.

Return values, page 444: *ERR_NO_RECIPE_MANAGER_SET, ERR_RECIPE_DEFINITION_NOT_FOUND, ERR_FAILED, ERR_OK*

Example:

There are 50 recipes. To create a table which shows 10 recipe names at a time, define an array of strings:

```
strArr: ARRAY[0..9] OF STRING;
```

Corresponding to the *iStartIndex*, the recipe names can be read from a specific area.

```
iStartIndex := 0;
```

The names 0...9 are returned.

```
iStartIndex := 20;
```

The names 20...29 are returned. In this example:

```
iSize := 10;
```

GetRecipeValues

This method returns the recipe variable values from the corresponding recipe.

Parameter	Data type	Description
RecipeDefinitionName:	STRING	Name of the recipe definition.
RecipeName :	STRING	Name of the recipe.
pStrings :	POINTER TO ARRAY [] OF STRING	The strings where the recipe values are to be stored.
iSize :	INT	The size of an array of strings.
iStartIndex :	INT	The start index; can be used for a scrolling function.
iStringLength :	INT	The length of the string in the array.

Return values, page 444: *ERR_NO_RECIPE_MANAGER_SET, ERR_RECIPE_DEFINITION_NOT_FOUND, ERR_RECIPE_NOT_FOUND, ERR_FAILED, ERR_OK*

Example:

There are 50 recipes. To create a table which shows 10 recipe names at a time, define an array of strings:

```
strArr: ARRAY[0..9] OF STRING;
```

Corresponding to the *iStartIndex*, the recipe names can be read from a specific area.

```
iStartIndex := 0;
```

The values 0...9 are returned.

```
iStartIndex := 20;
```

The values 20...29 are returned. In this example:

```
iStringLength := 80;
iSize := 10;
```

GetRecipeVariableNames

This method returns the variable name of the corresponding recipe.

Parameter	Data type	Description
RecipeDefinitionName:	STRING	Name of the recipe definition.
RecipeName :	STRING	Name of the recipe.
pStrings :	POINTER TO ARRAY [] OF STRING	The strings where the recipe values should be stored.
iSize :	INT	The size of an array of strings.
iStartIndex :	INT	The start index; can be used for a scrolling function.

Return values, page 444: *ERR_NO_RECIPE_MANAGER_SET, ERR_RECIPE_DEFINITION_NOT_FOUND, ERR_RECIPE_NOT_FOUND, ERR_FAILED, ERR_OK*

Example:

There are 50 recipes. To create a table which shows 10 recipe names at a time, define an array of strings:

```
strArr: ARRAY[0..9] OF STRING;
```

Corresponding to the *iStartIndex*, the recipe names can be read from a specific area.

```
iStartIndex := 0;
```

The names 0...9 are returned.

```
iStartIndex := 20;
```

The names 20...29 are returned. In this example:

```
iSize := 10;
```

SetRecipeValues

This method sets the recipe values into the corresponding recipe.

Parameter	Data type	Description
RecipeDefinitionName:	STRING	Name of the recipe definition.
RecipeName :	STRING	Name of the recipe.
pStrings :	POINTER TO ARRAY [] OF STRING	The strings where the recipe values should be stored.
iSize :	INT	The size of an array of strings.
iStartIndex :	INT	The start index; can be used for a scrolling function.

Return values, page 444: *ERR_NO_RECIPE_MANAGER_SET, ERR_RECIPE_DEFINITION_NOT_FOUND, ERR_RECIPE_NOT_FOUND, ERR_FAILED, ERR_OK*

Example:

There are 50 recipes. To create a table which shows 10 recipe names at a time, define an array of strings:

```
strArr: ARRAY[0..9] OF STRING;
```

Corresponding to the *iStartIndex*, the recipe names can be read from a specific area.

```
iStartIndex := 0;
```

The values 0...9 are set.

```
iStartIndex := 20;
```

The values 20...29 are set. In this example:

```
iStringLength := 80;
iSize := 10;
```

GetLastError

This method of data type DWORD returns the last detected error of the previous operations.

Return values, page 444: *ERR_NO_RECIPE_MANAGER_SET, ERR_OK*

ResetLastError

This method of data type DWORD resets the last detected error.

Return values, page 444: *ERR_NO_RECIPE_MANAGER_SET, ERR_OK*

SetStoragePath

This method allows you to set the storage path for the recipe file. It overwrites the **File Path** configured in the **Recipe Manager** dialog box, page 435.

Parameter	Data type	Description
stPath	STRING	The method returns TRUE if the path is set, as for example: D:/recipefiles/. The method returns FALSE: • If there is no **Recipe Manager** available in the application. • If there is no recipe definition, page 439 available below the **Recipe Manager** node.

Return values, page 444: *ERR_NO_RECIPE_MANAGER_SET, ERR_OK*

Loading Recipe Values from the Controller

Overview

You can apply recipe values from the controller to recipe definitions in the project, even if these definitions have been modified in the project.

As a prerequisite, the option **Recipe Management in the PLC** has to be activated in the **General** tab of the **Recipe Manager** editor and **File Path** has to be set to `'Recipes\'`.

Loading Recipe Values from the Controller

To load recipe values from the controller, proceed as follows:

Step	Action	Comment
1	Create a recipe definition **RecDef1** in the project, containing the variables *PLC_PRG.ivar* and *PLC_PRG.bvar*.	–
2	Insert a recipe R1: Value for *PLC_PRG.ivar*: `33` Value for *PLC_PRG.bvar*: `TRUE`	–
3	Log in to the controller and download the application.	**Result**: The recipe file R1.RecDef1.txtrecipe is saved to the directory **Recipes** of the controller.
4	Log out from controller.	–
5	Add another variable *PLC_PRG.dwvar* to the recipe definition in the project.	–
6	Edit the recipe definition file R1.RecDef1.txtrecipe on the controller by changing the value for *PLC_PRG:ivar* from `33` to `34`.	–
7	Add another recipe on the controller by copying the R1.RecDef1.txtrecipe file and renaming it to R2.RecDef1.txtrecipe.	–
8	Edit the R2.RecDef1.txtrecipe file and modify the recipe values: *PLC_PRG:ivar*: `1` *PLC_PRG.bvar*: `FALSE`	**Result**: On the controller, there are now two recipes, R1 and R2. In the project, there is only R1 available. This recipe R1 contains other values than R1 on the controller.
9	Log in to the controller.	**Result**: A dialog box opens.

Step	Action	Comment
10	From the dialog box, select the option **Login with online change** or **Login with download**, and click **OK**	–
11	While the recipe definition editor view is open, execute the command **Upload recipes from device** from the contextual menu of the **Recipe definition editor view**.	**Result**: A dialog box indicates that the recipe for PLC_PRG.dwvar loaded from the controller cannot provide a value from the controller.
12	Confirm the dialog box to continue.	**Result**: The value of *PLC:_PRG.ivar* in recipe R1 of the recipe definition in the project changes to `34`. *PLC_PRG.dwvar* remains in the recipe definition.

Memory Usage by Recipes

Memory Usage

If the option **Recipe Management in the PLC** is activated in the **General** tab of the recipe manager editor, then code is generated for the recipe manager and the recipe definitions. This code is saved in the controller. The size of the memory used depends on the number of recipes and their variables, as well as the data type of the variables. If the fields of the recipe definition are filled, this also requires memory space.

You can use the values provided in the table as a guideline for determining the memory space:

–	Code size (bytes)	Data size (bytes)	Total (bytes)
Recipe definition with 100 INT variables	194406	79400	267352
Recipe definition with 200 INT variables	238318	121284	459344
Recipe definition with 300 INT variables	282230	163084	543856
Recipe definition with 100 BOOL variables	192742	69884	343168
Recipe definition with 200 BOOL variables	235446	101568	436872
Recipe definition with 300 BOOL variables	278146	133284	510072
Recipe definition with 100 STRING variables	203278	870084	1154000
Recipe definition with 200 STRING variables	255570	1709784	2973296
Recipe definition with 300 STRING variables	307886	2549484	2964112

Trace Editor

What's in This Chapter

Trace Object 454
Trace Configuration 460
Trace Editor in Online Mode 469
Keyboard Operations for Trace Diagrams 471

Trace Object

Trace Basics

Trace Functionality

The trace functionality allows you to capture the progression of the values of variables on the controller over a certain time, similar to a digital sampling oscilloscope. Additionally, you can set a trigger to control the data capturing with input (trigger) signals. The values of trace variables are steadily written to a buffer of a specified size. They can be observed in the form of a two-dimensional graph plotted as a function of time.

NOTE: The tracing of data is continued when you log off the controller.

If PC processor-consuming tasks are executed on the PC running EcoStruxure Machine Expert while a trace is running, it may happen that some of the variable values are not captured by the trace.

NOTICE
LOSS OF DATA Avoid executing actions and/or PC applications that lead to a high CPU load while running a trace. **Failure to follow these instructions can result in equipment damage.**

Way of Tracing Data

The tracing of data on the control is performed in two different ways:

- Either from IEC code generated by the trace object and downloaded to the controller by a trace child application.
- Or within the *CmpTraceMgr* component (also named **Trace Manager**).

  NOTE: The tracing of data can significantly increase the cycle time of the IEC task.

What data is captured is determined by an entry in the target settings (**trace > trace manager**).

The trace manager has advanced functionality. It allows you to:

- Configure and trace parameters of the control system, such as the temperature curve of the processor or the battery. For more information, refer to the variable settings, page 462 and to the record (trigger) settings, page 464.
- Read out device traces, such as the trace of the electric current of a drive. For more information, refer to the description of the **Upload Trace** command (see Menu Commands, Online Help).
- Trace system variables of other system components.

Furthermore, the additional command (see Menu Commands, Online Help) **Online List** is available.

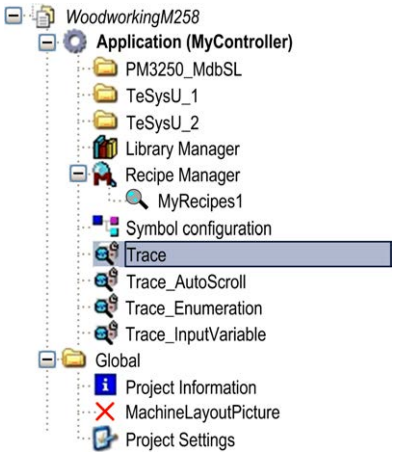
NOTE:

- If a trace is used in the visualization, device parameters cannot be traced or used for the trigger.
- The trigger level cannot be set to an IEC expression, only literals and constants are supported.
- The record condition cannot be set to an IEC expression of type BOOL, only variables are supported.
- If a property is traced or used for the trigger, it must be annotated with the attribute monitoring, page 615 in IEC declaration.

Configuration

Configure the trace data as well as the display settings of the trace data in the **Configuration**. It provides commands for accessing the configuration dialog boxes. Several variables can be traced and displayed at the same time, in different views such as multi-channel mode. Record traces of variables with different trigger settings in their own trace object. You can create any number of trace objects.

Tools tree with several trace objects



Commands for modifying the settings of the display are described in the *Features* paragraph, page 458. Zooming functionalities and a cursor are available as well as commands for running the trace so that the graph can be compressed or stretched.

To integrate the readout of a trace within a visualization, use the visualization element **Trace**.

For further information on trace configuration for data recording, refer to the *Record Settings*, page 464.

Creating a Trace or a Device Trace Object

Overview

To insert a trace object in the **Tools tree**, select the **Application** node, click the green plus button, and execute the command **Trace...**. Double-click the **Trace** node in the **Tools tree** to open the trace editor. **Trace** and **DeviceTrace** objects display trace data in one or more diagrams.

A **DeviceTrace** node, in particular, is inserted below the device in the **Devices tree** and does not immediately depend on the applications of the project. It directly accesses traces that are running on the controller.

Trace Variable Tree

The trace variable tree provides an overview of the trace configuration. The diagrams with the respective trace variables are displayed. Double-click a trace variable to open the **Trace Configuration** dialog box with the variable settings.

Click the triangle and select **Hide Instance Paths**. The variable is displayed in the tree without the full instance path. Example: `iCounter`

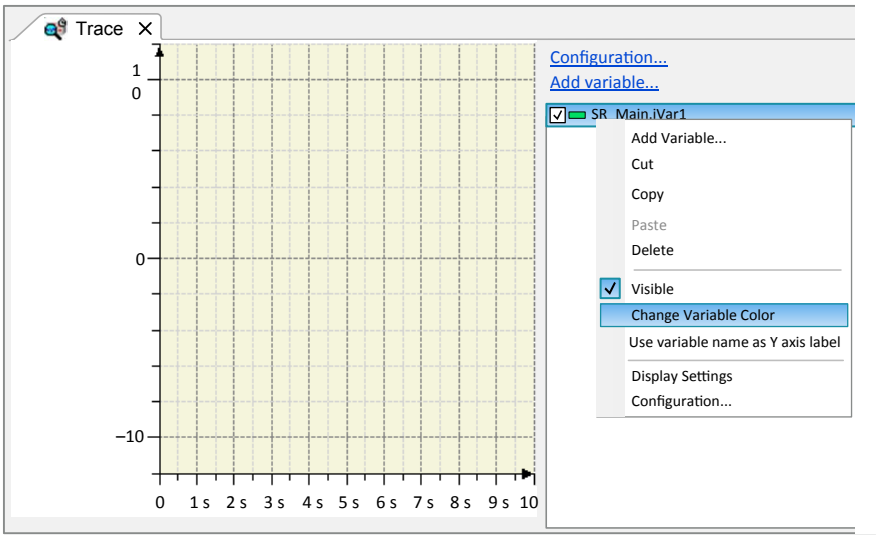
The tree view of the diagrams provides the following information:

Element	Description
Name	Diagrams are listed with the respective variables: • Select **Diagram <n>** to display the diagram. You can rename the diagram by editing the diagram name. • Select **<variable>** to display the variable. You can rename the variable by editing the variable name. A **Diagram <n>** selected in the tree is also selected in the editor and vice versa.
Cursor <n>	Indicates the Y value at the cursor position.
Delta	Indicates the delta of the Y value between **Cursor 1** and **Cursor 2**.
Multiselection	Diagrams selected simultaneously can be expanded and collapsed with the number pad keys ***** and **/**.

You can drag the diagrams and variables to sort them or move them to other diagrams. Hold down the **Ctrl** key to copy a variable. This is also available in online mode.

Configuration

Newly created trace with contextual menu:



A trace contains at least one variable which is sampled.

In the trace tree area in the right part of the window, the configured trace variables are displayed. By default, the trace variables are displayed with their complete instance path.

Select the **Hide instance paths** check box to hide the instance path. To display this check box, click the arrow button in the upper right corner of the trace tree area.

Select the **Hide diagrams** check box to hide the diagrams in the trace tree area.

To configure or modify the trace settings, use the commands of the contextual menu in the trace tree area:

- **Add Variable...**: Opens the **Trace Configuration** dialog box with **Variable Settings**, page 462.
- **Delete**: Deletes the selected variable. Only available if at least one trace variable exists.
- **Visible**: This command makes the selected variable visible. Only available if at least one trace variable exists.
- **Configuration...**: Opens the **Trace Configuration** dialog box with **Record Settings**, page 464.
- **Display Settings...**: Opens the **Display Mode** view, page 466. It allows you to configure the appearance of the graph and the coordinate system. This command is unavailable until a configuration is loaded.

Features

For running the trace, use the following commands:

- **Add variable** (see Menu Commands, Online Help)
- **Download Trace** (see Menu Commands, Online Help)
- **Start/Stop Trace** (see Menu Commands, Online Help)
- **Reset Trigger** (see Menu Commands, Online Help)

For customizing the view of the graphs, use the following commands:

- **Cursor** (see Menu Commands, Online Help)
- **Mouse Zooming** (see Menu Commands, Online Help)
- **Reset View** (see Menu Commands, Online Help)
- **Auto Fit** (see Menu Commands, Online Help)
- **Compress** (see Menu Commands, Online Help)
- **Stretch** (see Menu Commands, Online Help)
- **Move all variables to individual diagrams** (see Menu Commands, Online Help)
- **Move all variables to first diagram** (see Menu Commands, Online Help)
- For further information, refer to the chapter, page 471 *Keyboard Operations for Trace Diagrams*.

For access to traces stored on the runtime system, use the following commands:

- **Online List** (see Menu Commands, Online Help)
- **Upload Trace** (see Menu Commands, Online Help)

For access to traces stored on the disc, use the following commands:

- **Save Trace...** (see Menu Commands, Online Help)
- **Load Trace...** (see Menu Commands, Online Help)
- **Export symbolic trace config** (see Menu Commands, Online Help)

Getting Started

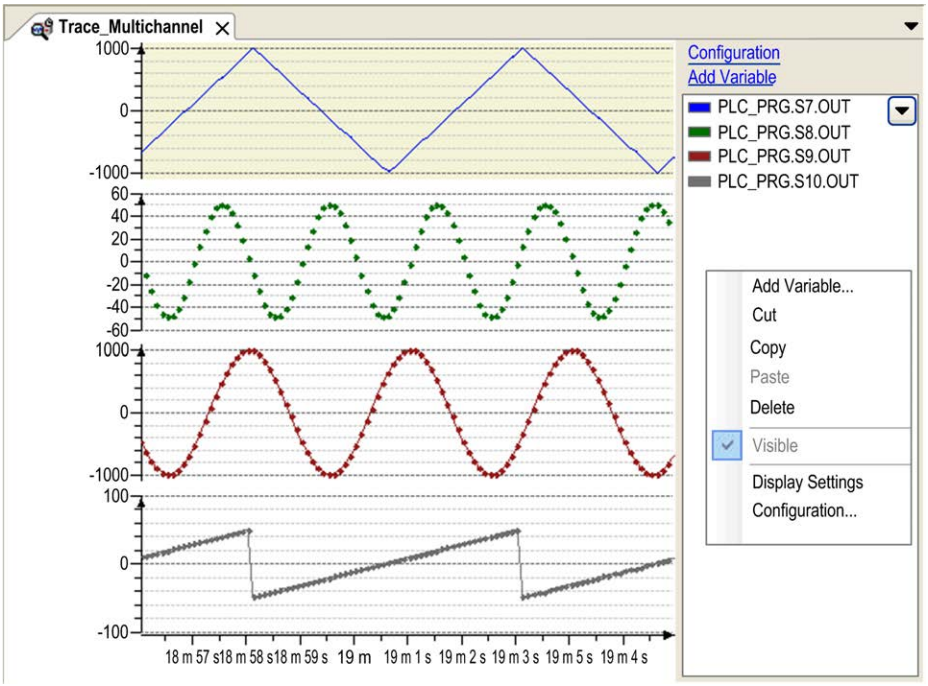
In order to start the trace in online mode, download the trace configuration to the controller by executing the **Download Trace** command. The graphs of the trace variables are displayed in the trace editor window where you can store them to an external file. This file can be reloaded to the editor. Also refer to the chapter *Trace Editor in Online Mode*, page 469.

Step	Action
1	Login and run the associated application. **Result:** The application runs on the controller.
2	Download trace **Result:** The trace graphs are immediately displayed according to the trace configuration.
3	Arrange the trace graphs, store the trace data, stop/start tracing.

Example

The trace editor shows an example of tracing in online mode. Four variables have been selected for display in the variables tree in the right part of the dialog.

Trace in online mode



Trace Configuration

Trace Configuration - Tree Views

Overview

The **Trace Configuration** dialog box provides two different tree views on the left-hand side:

- **Trace Record** tree view
- **Presentation (diagrams)** tree view

Trace Record Tree View

The **Trace Record** tree view lists the variables that are traced and provides access to the variable settings.

The view that is displayed on the right-hand side of the **Trace Configuration** dialog box depends on the element that is selected in the tree view:

If...	Then ...
A trace name is selected	The **Record Settings** view is displayed on the right-hand side.
A trace variable is selected	The **Variable Settings** view is displayed on the right-hand side.

Commands of the Contextual Menu of the Trace Record Tree View

The contextual menu of the **Trace Record** tree view provides the following commands:

Command	Description
Add variable	Opens the **Variable Settings** view, page 462 on the right-hand side. To add a new variable, click the browse button (**...**), and select a variable for tracing.
Assign to diagram	Right-click a variable and execute the **Assign to diagram** command to list the diagrams that do not include the selected variable. Select a diagram to display the selected variable in this diagram. This command is deactivated if the selected variable is already assigned to all diagrams.
Enabled	The variables are by default enabled. Deactivated variables are displayed as disabled. They are not displayed or recorded.

Presentation (diagrams) Tree View

The **Presentation (diagrams)** tree view lists the diagrams that are displayed in the trace editor and provides access to the display mode.

The view that is displayed on the right-hand side of the **Trace Configuration** dialog box depends on the element that is selected in tree view:

If...	Then ...
The node **Time axis** is selected.	The **Display mode** view, page 466 is displayed on the right-hand side. It allows you to specify the time axis display.
A diagram name is selected	The settings for the coordinate system of the diagram and a preview are displayed.
The node **Y axis** is selected.	The **Display mode** view, page 466 is displayed on the right-hand side. It allows you to specify the axis display.
A trace variable is selected below the **Shown variables** node.	The **Variable Settings** view, page 462 on the right-hand side that allows you to configure the trace variables.

Commands of the Contextual Menu of the Presentation (diagrams) Tree View

The contextual menu of the **Presentation (diagrams)** tree view provides the following commands:

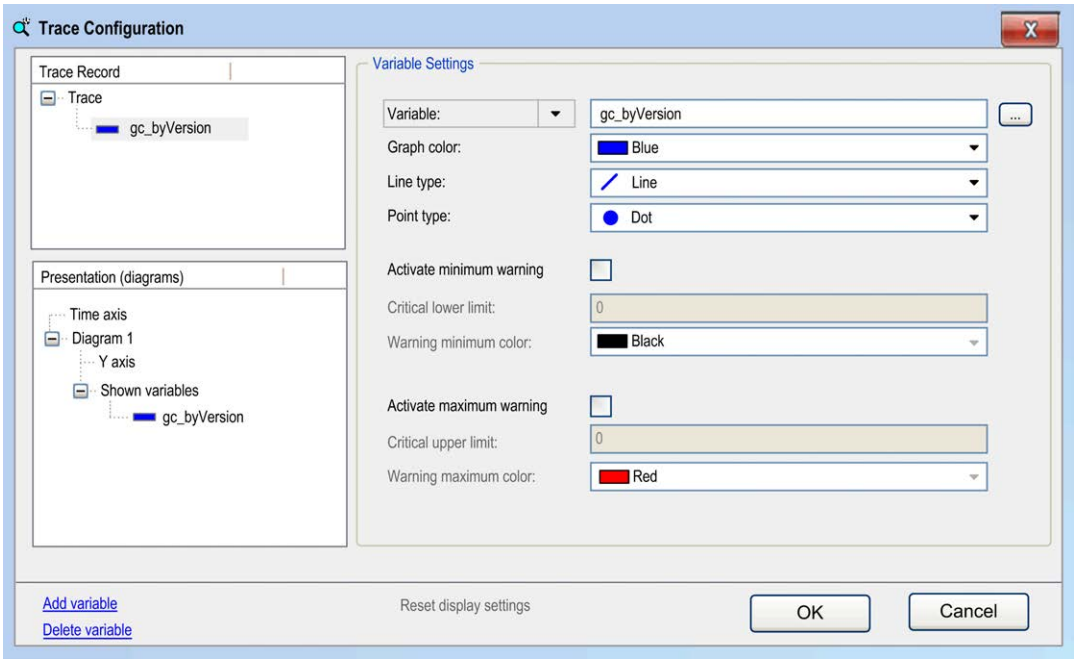
Command	Description
Add diagram	Adds a new diagram and displays it in the **Presentation (diagrams)** tree view.
Add new variable	Adds a new trace variable and opens the **Variable Settings** view, page 462 on the right-hand side. Select a variable in the input field to trace the value graph. Specify the display. The variable is assigned to the selected diagram.
Add existing variable	Right-click a diagram and execute the **Add existing variable** command to list the trace variables where the selected diagram is not displayed. Select a variable to display it in the selected diagram. This command is not available if all trace variables are already displayed in the selected diagram.

Variable Settings

Overview

The **Trace Configuration** dialog box with **Variable Settings** opens when you select a trace variable in the trace tree. It allows you to configure which variables should be traced and how they are displayed.

Trace Configuration dialog box with **Variable Settings**



The trace variables are displayed in the left part of the window in a tree structure. The top node is titled with the trace name.

Adding and Deleting a Trace Variable

For adding a variable to the trace tree or deleting one, use the commands below the trace tree:

Command	Description
Add Variable	Creates an anonymous entry in the trace tree. In the right part of the dialog box, the settings of the new variable are ready for configuration.
Delete	Deletes the selected variable with the associated configuration.

Setting and Modifying the Variable Settings

To choose the variable settings, select the desired variable in the trace tree. The present settings will be displayed in the right part of the trace configuration window. To modify the variable settings later, select the variable entry in the trace tree and use the **Variable Settings** dialog box again.

Parameter		Description
Variable		Enter the name (path) of the signal to specify the signal that is traced. A valid signal is an IEC variable, a property, a reference, the contents of a pointer, or an array element of the application. Allowed types are all IEC basic types, except STRING, WSTRING or ARRAY. Enumerations are also allowed, whose basic type is not a STRING, a WSTRING or an ARRAY. Click the **...** button to open the input assistant that allows you to obtain a valid entry. Controllers that support parameter tracing provide a list if you click the **Variable:** parameter. If you want to trace a device parameter, select the item **Parameter** from this list. Then you can find one with the help of the input assistant. Edit or verify the variable settings. Device parameters are only supported if the *CmpTraceMgr* component is used. If device parameters are used for trace (or for trigger) variables, it is not possible to activate the option **Generate Trace POU for visualization**. **NOTE:** If *CmpTraceMgr* is used for tracing, a Property, page 184 that is used as a trace (or trigger) variable must get the compiler attribute Attribute Monitoring, page 615. **NOTE:** Do not use Sercos parameters of type *AS* for tracing as this can reduce the performance of your system.
Graph color		Select a color from the color selection list in which the trace curve for the variable is displayed.
Line type		Specify the way samples are connected in the graph. Use **Line** for large volumes of data. It is also the default value.
	Line	The samples are connected to a line (default).
	Step	The samples are connected in the shape of a staircase. Thus, a horizontal line to the time stamp of the next sample followed by a vertical line to the value of the next sample.
	None	The samples are not connected.
Point type		Specify how the values themselves are drawn in the graph.
	Dot	The samples are drawn as dots (default).
	Cross	The samples are drawn as crosses.
	None	The samples are not displayed.
Activate Minimum Warning		If this option is activated, the trace graph is displayed in the color defined in **Warning minimum color** as soon as the variable exceeds the value defined in **Critical lower limit**.
Critical lower limit		If the value of the variable entered here has fallen below and **Activate minimum warning** is active, the values of the curve changes in the following specified color.
Warning minimum color		Color value for the activated lower limit.

Parameter	Description
Activate Maximum Warning	If this option is activated, the trace graph is displayed in the color defined in **Warning maximum color** as soon as the variable exceeds the value defined in **Critical upper limit**.
Critical upper limit	If the value of the variable entered here is exceeded and **Activate maximum warning** is active, the values of the curve changes in the following specified color.
Warning maximum color	Color value for the activated upper limit.

Multi-Selection of Variables

By using the keyboard shortcuts **Shift** + mouse-click or **Ctrl** + mouse-click, you can select several variables for editing. Then the modifications made in the dialog box **Variable Settings** are applied to all selected variables. The same can be achieved with **Shift** + **Up Arrow**/**Down Arrow** or **Ctrl** + **Up Arrow**/**Down Arrow**.

Record Settings

Overview

The **Trace Configuration** dialog box with **Record Settings** opens if you execute the command **Configuration** or if you double-click a trace variable in the trace tree. The configuration command is also available in the contextual menu of the trace tree on the right part of the main trace editor window.

NOTE: The settings completed in the dialog box **Trace Configuration** dialog box with **Record Settings** are valid for all variables of the trace graph.

Trigger Basics

In most situations, it is not desired that tracing and displaying the input signals starts at random moments, such as immediately after the previous measurement, or when the user presses the start button. Most of the time it is preferred that the tracing is done when a trigger is fired for the configured number of records (post trigger). This is called triggering and has to be defined here.

The following ways are used for triggering input signals:

- by configuring a trigger variable
- by configuring a record condition
- or both

Setting and Modifying the Record (Trigger) Settings

Parameter	Description
Enable Trigger	Select the checkbox to enable the trigger system. It can be turned on or off independently of the lower settings. If the trigger system is disabled, the trace is free-running.
Trigger Variable	Assign a variable. Specify which signal will be used as trigger by entering the name (and path) of the signal. A valid trigger signal is an IEC variable, a property, a reference, a pointer, an array element of the application or an expression. Allowed types are all IEC basic types, except STRING, WSTRING, or ARRAY. Enumerations are also allowed, whose basic type is not a STRING, a WSTRING or an ARRAY. The content of a pointer is not a valid signal. Click the **...** button to open the input assistant that allows you to obtain a valid entry.

Parameter		Description
		Controllers that support using device parameters as triggers provide a list if you click the **Trigger Variable:** parameter. If you want to use a device parameter as trigger, select the item **Trigger Parameter** from this list. Open the **Input Assistant** with the **...** button and select traceable parameters. Under **Elements**, the parameters available in the system are listed. You can also type the parameter names directly or by copy and paste (from another configuration) into the text field. Device parameters are only supported if the trace manager is used. **NOTE:** If *CmpTraceMgr* is used for tracing, a Property, page 184 that is used as a trace (or trigger) variable must get the compiler attribute Attribute Monitoring, page 615.
Trigger edge		–
	positive	Trigger event on rising edge of the boolean trigger variable. Or as soon as the value defined by **Trigger Level** for an analog trigger variable is reached by an ascending run.
	negative	Trigger event on falling edge of the boolean trigger variable. Or as soon as the value defined by **Trigger Level** for an analog trigger variable is reached by a descending run.
	both	Trigger event on the conditions described for **positive** and **negative**.
Post Trigger		Enter a number of records per trace signal, which are recorded after the trigger is fired. Default value: 50 Range: 0...4,294,967,295
Trigger Level		Enter a value at which point the trigger fires. With **Trigger edge**, you can specify whether it fires on a rising or a falling edge of the trigger variable. It must be set if and only if an analog variable (variable with numeric type, such as LREAL or INT) is used as trigger variable. Directly enter a value. A GVL constant or an ENUM value is allowed if their type is convertible to the trigger variable. If IEC code is used, then you can also enter an arbitrary IEC expression of a type that is convertible to that of the trigger variable. Default value: – (empty)
Task		From the list of available tasks, select the one where capturing of the input signals takes place.
Record condition		If you want to start the record by a condition, enter here a variable. If the trace is started before, for example by pressing the start button, and the variable assigned here becomes TRUE, the data capturing is started and the traced graph will be displayed. If *CmpTraceMgr* is used, the record condition has to be a variable of type BOOL or of a bit-access. A content of a pointer is not a valid entry. Properties are also supported. If IEC code is used, an arbitrary IEC expression of type BOOL can also be entered.
Comment		Enter a comment text concerning the present record.
Resolution		Enter a resolution of the trace time stamp in ms or µs. For each captured signal, pairs of value and time stamp are stored and transmitted to the programming system. The transmitted time stamps are relative and refer to the start of the tracing. If the trace task has a cycle time of 1 ms or less, use a time stamp with resolution in µs. This option is only possible when a trace manager is available in the controller.

Parameter	Description
Automatic restart	Set this option if the trace configuration and the last content of the RTS trace buffers is to be stored persistently on the target device. This option is only possible when a trace manager is tracing in the controller.
Advanced...	Click this button to open the **Advanced Trace Settings** dialog box, page 468. It allows you to set some additional settings for the trace trigger.

NOTE: If you want to capture and display a trace signal with a different time base, you have to do a record configuration in a separate trace object.

Display Mode

Overview

The **Display mode** view opens on the right-hand side when you select an axis (time or Y axis) in the **Presentation (diagrams)** tree view.

The parameters displayed depend on the axis that is selected:

Parameter	Description
Display mode	Select the following scaling options: • **Auto** If this option is selected, the axis is scaled. • **Fixed length** Only available for time axis: If this option is selected, a segment of constant length is displayed. • **Fixed** If this option is selected, a time or Y axis defined by the **Minimum** and **Maximum** parameters is displayed.
Minimum	When the **Display mode > Fixed** is selected, enter a start value for the axis. It can be a literal, variable (integer data type), or constant variable (integer data type).
Maximum	When the **Display mode > Fixed** is selected, enter an end value for the axis.
Length	Only available for a time axis when the **Display mode > Fixed length** is selected. Enter a value to define a constant length for the axis. The start value is adapted accordingly.
Grid	Select the option **Grid** to create a diagram with grid lines in x- or y-direction. Select the color of the grid lines from the list next to the **Grid** parameter.
Description	Only available for Y axis. Select the option **Description** and enter a describing text in the box that will be displayed in the upper left corner of the diagram.
Tick marks section	
Fixed spacing	If the option **Fixed spacing** is selected, tick marks with **Distance** and **Subdivisions** are drawn.
Distance	Enter a value to define the distance between the tick marks.
Subdivisions	Enter a value to define the number of subdivisions between two tick marks.

The following elements are available:

Element	Description
Font button	Click the **Font** button to open the **Font** dialog box and configure the font for the selected axis.
Reset button	Click the **Reset** button to reset the settings to the default values.
Use as default button	Click the **Use as default** button to save the settings as default values.
Preview link	Click the **Preview** link to see a preview of the diagram you defined in the **Display mode** view.

If a trace is selected in the **Trace Record** tree view, the following elements are available:

Element	Description
Add variable link	Click the link to add a new trace variable to the **Trace Record** tree view.
Delete variable link	Click the **Delete variable** link to delete the selected trace variable from the **Trace Record** tree view.

Preview

The preview of the diagram you defined in the **Display mode** view is displayed when you click the **Preview** link in the **Display mode** view.

Underneath the preview of the diagram, the following elements are available:

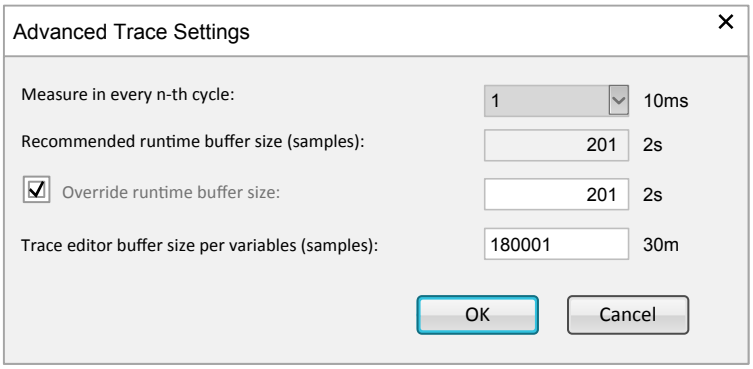
Element	Description
Backcolor parameter	Select a color for the background of the diagram from the list next to the **Backcolor** parameter.
Backcolor on Selection parameter	Select a color for the background of the selected diagram from the list next to the **Backcolor on selection** parameter.
Add diagram link	If a diagram is selected, click the **Add diagram** link to add a new diagram to the **Display mode** view.
Delete diagram link	If a diagram is selected, click the **Delete diagram** link to delete the selected diagram from the **Display mode** view.
Reset display settings link	Click the **Reset display settings** link to reset the display settings of the selected diagram or the selected y-axis to the default values.
OK button	Click the **OK** button to confirm the configuration settings and to save the trace configuration.

Advanced Trace Settings

Overview

The **Advanced Trace Settings** dialog box opens if you click the **Advanced...** button in the **Trace Configuration** dialog box, page 464 with **Record Settings**. It includes settings for the data recording, such as the size of the ring buffer on the runtime or development system that the trace editor reads. The interval that the application uses to record data is derived from the task configuration.

Advanced Trace Settings dialog box



You configure the buffer size by defining a number of samples. Using settings from the task configuration, time intervals according to the number of samples are calculated. The results are displayed on the right-hand side of each parameter.

NOTE: For the calculation to be performed, a task must be selected and the settings of the trace configuration must be set in the **Trace Configuration** dialog box.

Description of the Parameters

Parameter	Description	Comment
Measure in every n-th cycle	Defines how often a data record is taken (every cycle, every second cycle,...). This is the scanning rate. Default: 1 Therefore, one data set is recorded each cycle.	For example, the scanning interval of the data recording is every 10 ms.
Recommended runtime buffer size (samples)	The maximum number of samples that are calculated and that the runtime system can store per trace variable. The number is calculated in the task cycle time from the value in **Refresh interval** and the value in **Measure in every n-th cycle**. The **Recommended runtime buffer size (samples)** is only used if the option **Override runtime buffer size** is not activated.	Maximum length of the time interval during which the runtime system collects data, for example, 2 s (if **Refresh interval** = **500 ms** and **Measure in every n-th cycle** = **1**).
Override runtime buffer size	When this option is selected, the application does not use value set for **Recommended runtime buffer size (samples)**. The number of samples per trace variable that the application records. This is the size of the runtime buffer. Range: From 10 on, but not larger than the value set for the trace editor buffer. Default value: 100	Maximum length of the time interval during which the runtime system collects data, for example, 6 s.
Trace editor buffer size per variables (samples)	Limits the trace buffer provided by EcoStruxure Machine Expert for internal use.	–

Trace Editor in Online Mode

Trace Editor in Online Mode

Overview

If a trace is running on the device, it is indicated in the trace dialog box **Online List** (see Menu Commands, Online Help).

Download Trace

In order to start the trace in online mode, download explicitly the trace to the controller with the **Trace > Download Trace** menu command (see Menu Commands, Online Help) while the application is logged in. The graphs of the trace signals will be displayed in the trace editor window.

While doing logins and logouts on the application without changing it, the traces are running without a new download.

If the application code is changed, then it depends on the login mode, in what happens with the traces:

- **Login with online change** or **Login without any change**: The traces are still running.
- **Login with download**: The traces in the controller are deleted and a new download of them is necessary.

Online Change of the Trace Graph Configuration

The **Trace Configuration** dialog box with **Record Settings** and the **Trace Configuration** dialog box with **Variable Settings** are available in online mode and quite a few changes on the trace configuration can be performed while the trace is running. If this is not possible, when the name of the trace signal is changed; for example, the trace is stopped and a new download is required.

Online Navigation of the Trace Graph

The displayed range of the captured trace variable values depends not only on the trace configuration. It can also be rearranged by scroll and zoom functionalities available in the **Trace** menu, the toolbar or by using shortcuts. For information on how to navigate in the trace diagram, refer to the *Keyboard Shortcuts* chapter, page 471.

Keyboard Operations for Trace Diagrams

Keyboard Shortcuts

Overview

The following table describes keyboard and mouse actions:

Actions	By Keyboard Operation	By Mouse Operation
Scroll the trace graph horizontally along the time axis.	No trace cursor: • ARROW LEFT/RIGHT • With greater distances: CTRL + ARROW LEFT/ RIGHT 1 or 2 trace cursors: • ALT + ARROW LEFT/ RIGHT • With greater distances: CTRL + ALT + ARROW LEFT/RIGHT	Scroll the graph by drag and drop. This is indicated by a different view of the mouse cursor.
Scroll the trace graph vertically along the Y-axis.	ARROW UP/DOWN With greater distances: CTRL + ARROW UP/DOWN	Use CTRL + drag and drop.
Zoom to a rectangle (window) that is selected with the mouse.	–	Use the command **Mouse Zooming** (see Menu Commands, Online Help).
Shift the black trace cursor.	ARROW LEFT/RIGHT With greater distances: CTRL + LEFT/RIGHT ARROW	Click the black triangle of the trace cursor, drag it along the X-axis until you drop it.
Shift the gray trace cursor.	SHIFT + ARROW LEFT/ RIGHT With greater distances: CTRL + SHIFT + ARROW LEFT/ RIGHT	Click the gray triangle of the trace cursor, drag it along the X-axis until you drop it.
Compress the time axis. In multi-channel mode, the time axes for all diagrams are compressed.	–	Use the mouse wheel. Or use the command **Compress** (see Menu Commands, Online Help).
Stretch the time axis. In multi-channel mode, the time axes for all diagrams are stretched.	+	Use the mouse wheel. Or use the command **Stretch** (see Menu Commands, Online Help).
Compress the Y-axis. In multi-channel mode, the Y-axis for the selected diagrams is compressed.	CTRL + –	CTRL + mouse wheel
Stretch the Y-axis. In multi-channel mode, the Y-axis for the selected diagrams is stretched.	CTRL + +	CTRL + mouse wheel
Selection of the next diagram below in multi-channel mode.	TAB	Click an unselected diagram to select this one.

Trend Recording

What's in This Chapter

Trend Recording Objects 472
Trend Recording Configuration 474
Recording Trend Data 480

A trend element can be added to a visualization. For details regarding the configuration of trend visualization, refer to *Displaying Data Graphs with a Trend Element* in the online help under **Software > Programming and Configuring > CODESYS Visualization > Recording Data and Displaying as a Trend**.

NOTE: A trend element can only be configured if a **Webvisualisation** object has been added to the **Visu Manager** object.

Trend Recording Objects

Trend Recording Overview

Trend Recording Functionality

Trend recording offers a long-term acquisition of data. With the help of a trend recording, values of IEC variables are acquired and saved in a database. Historical data are retained so that a trend can be read from the data. The database is located on the controller and records are written during runtime. Trend data can be displayed by a visualization, which reads out the database and visualizes the records.

Depending on the amount of data that is recorded and the performance of the controller, it may happen that a timeout expires during trend recording operations. This is detected as an exception error.

Before you stop the controller or perform a download (execute the **Online change** command (see Menu Commands, Online Help)), stop the trend recording process to save the cached trend data to the database.

NOTICE
LOSS OF DATA Stop the trend function and save the trend data before changing state of the controller from the run state. **Failure to follow these instructions can result in equipment damage.**

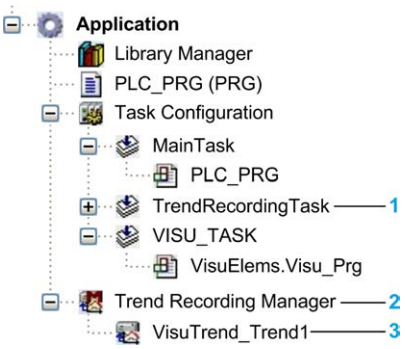
This function is not available for all supported controllers. Consult the *Programming Guide* specific to your controller as to whether it supports trend recording.

Trend Recording Objects

Overview

To insert a trend recording object in the **Tools tree**, select the **Application** node. Add a visualization and a trend visualization element. This function is not available for all controllers. Consult the *Programming Guide* specific to your controller for further information. In this example, a **Trend Recording Manager** and an object of the type a **Trend Recording** are added as subnodes to the **Application** node.

Tools tree with trend recording objects:



1TrendRecordingTask of the type **Task**

2 Trend Recording Manager

3 Object <name of the trend recording> of the type **Trend Recording**

Trend Recording Task Object

The **Trend Recording Task** object extends an application by a task in which the **Trend Recording Manager** is executed. In the **Tools tree**, only one **Trend Recording Manager** subnode is available per **Application** node. The task calls the following program: `VisuTrendStorageAccess.GlobalInstances.g_TrendRecordingManager.CyclicCall`.

If you add a trend in a visualization, then the **Task configuration** node is extended by a **Trend Recording Task** object.

Trend Recording Manager Object

The **Trend Recording Manager** object extends an application by the long-term data storage function with the help of a database. In the **Tools tree**, only one **Trend Recording Manager** subnode is available per **Application** node.

NOTE: Visualization elements for trends are supported in web visualizations only. They are no longer supported in integrated visualizations.

Object of the Type Trend Recording

An object of the type **Trend Recording** extends an application by the function **Place data in a database**. The data is acquired with the trace function. Below an **Application** node, any number of trend recording objects is allowed. Configure trend recording in the **Trend Recording Editor**.

Trend Recording Configuration

Trend Recording Editor

Overview

Double-click a trend recording object in the **Tools tree** or execute the command **Visualization > Edit trend recording** to open the **Trend Recording Editor**.

The editor contains a tree view of the configuration and displays the **Record Settings** or the **Variable Settings**, depending on the selection in the tree view.

Tree View of the Configuration

The tree view of the trend shows the trend variables that are configured and enables navigation to the settings. The top node contains the name of the trend recording. Clicking this entry opens the associated **Record Settings**. Under the top node, there are nodes for each trend variable whose values are acquired. Clicking an entry opens the associated **Variable Settings**.

Record Settings

Overview

The **Trend Recording** dialog box with **Record Settings** opens when you select the top node in the tree view of the **Trend Recording Editor**.

The recording of data is performed using the function of the runtime component that is also used for tracing, page 464. The settings that appear here are the same. The parameters that are related to trigger settings are not required during trend recording and are therefore grayed out.

Setting and Modifying the Recording Settings

The **Trend Recording Configuration** dialog box with **Record Settings** contains the following configuration elements:

Element	Description
Task parameter	From the list, select the task in which data acquisition is executed. In general, the trend recording runs in the same task as the main program, for example, `PLC_PRG`.
Record condition parameter	Refers to a variable of the type BOOL, to a bit or to a property. The data acquisition is executed if the value is TRUE. The setting is optional.
Comment parameter	Enter a comment text.
Resolution parameter	Resolution with which the time stamp is acquired. Set the resolution of the time stamp to 1 µs if the trend task has a cycle time of 1 ms or less.
Trend storage... button	Click this button to open the **Trend storage** dialog box. It allows you to configure further parameters that concern the behavior of the trend recording at runtime.
Advanced... button	Click this button to open the **Advanced Trend Settings** dialog box, page 475. It allows you to configure parameters that concern the acquisition and storage of data.

Trend Storage Dialog Box

Click the **Trend Storage...** button in the **Record Settings** dialog box to open the **Trend Storage** dialog box.

The **Trend Storage** dialog box contains the configuration for buffering data during trend recording.

Parameter		Description
Maximum number of variables		Defines the maximum number of trend variables that are managed in the database. If you modify this value, the database on the controller is updated by forcing a download during next login to the controller. The database will then be uploaded from the controller, the variables of the database will be increased or decreased according to this new setting, and will be downloaded to the controller.
Store every N milliseconds		Number of ms defining the time interval at which the recorded data is temporarily stored in a local buffer before being stored persistently in the database. The time interval is converted internally to the number of task cycles. The duration of a task cycle is defined in the task configuration. A high value results in a better runtime performance. The disadvantage of this is that the greater the value, the amount of data subject to be lost in the case of a controller change of state is likewise increased. A low value reduces this risk. The disadvantage of this is that the control of a trend visualization is degraded with large amounts of data.
Limit		Activate an option to limit the recording.
	No Limit	The amount of trend recording data that is stored is not limited. Use this option only for testing purposes, as using this setting will eventually exhaust the file space in the controller if not regularly maintained.
	Maximum number of records	This option defines the maximum number of data records that are stored in the database. Each data record consists of time stamp and the values of the trend variables. **NOTE:** If this limit is reached, trend recording is stopped.
	Maximum storage size	This option defines the maximum memory size reserved for trend storage. The number of records that can be stored within this memory depends on the size of the individual records. Select the suitable unit from the list (kilobytes (Kb), megabytes (Mb), or gigabytes (GB)). **NOTE:** If the memory reserved for trend storage is full, trend recording continues, overwriting the oldest records.

Advanced Trend Settings Dialog Box

Click the **Advanced...** button in the **Record Settings** dialog box to open the **Advanced Trend Settings** dialog box.

The **Advanced Trend Settings** dialog box enables the configuration of further parameters that concern the behavior of the trend recording at runtime.

Parameter	Description
Measure in every n-th cycle	Defines how often data is acquired by the runtime system, depending on the number of cycles completed. The time interval is calculated according to the frequency by using the settings from the task configuration. Thus, the calculation can only be performed if at least the task cycle time is set. The result is shown on the right of the input field, for example, **1h1m1s1ms**. The default setting is **1**. Therefore, data is acquired every cycle.
Additional Runtime Buffer for	Defines the time, in ms, that the runtime system can use an additional “overflow” buffer for recording data. Example: **1000 ms** **NOTE:** This additional buffer can help to avoid loss of data if a delay occurs during storage of the trend data.

Variable Settings

Overview

The **Trend Recording** dialog box with **Variable Settings** opens when you select a trend variable in the tree view of the **Trend Recording Editor**. It allows you to configure which variables and parameters should be recorded and how they are displayed.

The **Trend Configuration** dialog box with **Variable Settings** corresponds to the dialog box for tracing, page 462. The trend variables are displayed in the left part of the window in a tree structure. The top node is titled with the trend name.

Setting and Modifying the Variable Settings

To choose the variable settings, select the desired variable in the tree view on the left-hand side. The present settings are displayed in the right part of the dialog box. To modify the variable settings later, select the variable entry in the tree view and use the **Variable Settings** dialog box again.

Parameter	Description
Variable / Parameter	Enter the IEC variable from which values shall be acquired. The variable / parameter must have a valid type. A valid type is an IEC variable, a property, a reference, the contents of a pointer, or an array element of the application. Allowed types are all IEC basic types, except STRING, WSTRING or ARRAY. Enumerations are also allowed, whose basic type is not a STRING, a WSTRING or an ARRAY. Click the **...** button to open the **Input Assistant** that allows you to obtain a valid entry. Controllers that support trend recording provide a list if you click the **Variable:** parameter. If you want to create the trend of a device parameter, select the item **Parameter** from this list. Then you can find one with the help of the **Input Assistant**. Edit or verify the variable settings.
Attached y axis	Y-axis of the trend diagram that displays the **Variable**. The list provides the default Y-axis and the configured Y-axes. To configure already available y-axes, right-click the trend visualization element and execute the command **Configure Trend Display Settings...** from the contextual menu (or click the **Display settings** property from the **Properties** view).
Display variable name	If the option is selected, the name of the IEC variable is displayed by the visualization in the trend diagram in runtime mode. If the option is not selected, the **Description** is displayed.

Parameter	Description
Description	Enter text that is displayed instead of the variable name in the **Legend** assigned to the trend visualization element by having selected a **Legend** in the **Properties** view. This text is inserted in the *GlobalTextList* object where it can be localized. Example: `Sensor A (PLC_PRG.iSensor_A)` If you do not enter text in this **Description** box, the option **Display variable name** is selected by default, and only the name of the variable is displayed. Example: `PLC_PRG.iSensor_A` This is only applicable if a legend is assigned to the trend element.
Curve type	• **Line** • **Area**
Graph color	Color with which the variable is displayed in the trend diagram. Click  to open the color selection dialog box.
Line type	Defines how the acquired values are displayed as a line chart. Use the option **Line** for large amounts of data. As a prerequisite, the **Curve type** must be set to **Line**. Defines how the acquired values are displayed as a line chart: • **Line**:The values are linked to form a line - use this setting for large amounts of data. • **Step**:The values are linked in the form of steps. • **None**:The values are not linked.
Line width	Width of the line in pixels. Example: `1`
Line style	Select the display of the line: • **Solid** • **Dashes** • **Dots** • **Dash dot** • **Dash dot dot**
Point type	Defines how the acquired values are displayed as a dot chart: • **None**: No dot chart - use this setting for large amounts of data. • **Dot** • **Cross** • **Small circle** • **Big circle**
Filling type	As a prerequisite, the **Curve type** must be set to **Area**. Select the filling of the area: • **No filling** • **Plain color** • **Gradient**
Filling color	If selected, the area is filled with the selected color. As a prerequisite, the **Curve type** must be set to **Area**.
Transparency	Enter a value between 0...255 for defining the transparency of the selected color. Examples: `255` = the color is opaque; `0` = The color is transparent. As a prerequisite, the **Curve type** must be set to **Area**.
Activate minimum warning	If this option is activated, a message is displayed when the variable value is under the lower limit.
Critical lower limit	If the value of the trend variable falls below the limit, the variable is displayed in the **Color** defined for values below the limit.
Warning minimum color	Color in which the variable is displayed when under the lower limit.

Parameter	Description
Activate maximum warning	If this option is activated, a message is displayed when the variable value is above the upper limit.
Critical upper limit	If the value of the trend variable exceeds the limit, the variable is displayed in the **Color** defined for values above the limit.
Warning maximum color	Color in which the variable is displayed when above the upper limit.

A valid trend variable can be an IEC variable, a property, a reference, the contents of a pointer, an ARRAY element, or an enumeration, as long as the type is not STRING, WSTRING, or ARRAY.

Steps for Trend Recording Configuration

Overview

Configure trend recording by performing the following steps:

- Assigning task.
- Inserting trend variable.
- Deleting trend variable.
- Configuring trend memory.
- Configuring advanced trend settings.

Assigning Task

Define the task in which the trend recording is to be executed. In general, the trend recording runs in the same task as the main program, for example, `PLC_PRG`.

Step	Action
1	Double-click a trend recording object in the **Tools tree**. **Result**: The **Trend Recording Editor** opens.
2	Select the top node in the tree view. **Result**: The **Record Settings** are displayed on the right-hand side of the dialog box.
3	Open the **Task** list and select the task in which the associated application runs.

Inserting Trend Variables

The number of trend variables is limited and is set in the **Trend storage** dialog box.

Step	Action
1	Double-click a trend object in the **Tools tree**. **Result**: The **Trend Recording Editor** opens.
2	Execute the **Add variable** command, which is located below the tree view. **Result**: A new variable is inserted and the corresponding variable settings are displayed.
3	Enter a valid IEC variable from the **Variable** list, or select the entry **Parameters** and enter a valid parameter. **Result**: A signal has been assigned to the trend variable.

Step	Action
4	Configure the display of the trend variable.
5	Configure the color of the trend variable.

Also refer to the **Variable Settings** of the **Trace Configuration**, page 462.

Deleting Trend Variables

Step	Action
1	Double-click a trend object in the **Tools tree**. **Result**: The **Trend Recording Editor** opens.
2	Click a trend variable in the tree view. **Result**: The variable is selected.
3	Execute the **Delete variable** command, which is located below the tree view or press the **Delete** key.

Configuring Trend Memory

Step	Action
1	Double-click a trend object in the **Tools tree**. **Result**: The **Trend Recording Editor** opens.
2	Select the top node in the tree view. **Result**: The **Record Settings** are displayed on the right-hand side of the dialog box.
3	Click the **Trend storage...** button. **Result**: The **Trend Storage** dialog box opens, page 475.
4	Adapt the settings.

Configuring Advanced Trend Settings

Step	Action
1	Double-click a trend object in the **Tools tree**. **Result**: The **Trend Recording Editor** opens.
2	Select the top node in the tree view. **Result**: The **Record Settings** are displayed on the right-hand side of the dialog box.
3	Click the **Advanced...** button. **Result**: The **Advanced Trend Settings** dialog box opens.
4	Adapt the settings.

Recording Trend Data

Start the Recording Process

Precondition

To start the recording process, a configured object of the type **Trend recording** is required.

Start Trend Recording

Step	Action
1	Start the controller.
2	Select the application that contains the trend recording and execute the command **Application > Login**.
3	Start the application.

Before you stop the controller or perform a download (execute the **Online change** command (see Menu Commands, Online Help)), stop the trend recording process to save the trend data to the database.

NOTICE
LOSS OF DATA Stop the trend function and save the trend data before changing state of the controller from the run state. **Failure to follow these instructions can result in equipment damage.**

Conditionally Start Trend Recording

Precondition: Trend configuration has been carried out.

Step	Action
1	Double-click a trend object. **Result**: The **Trend Recording Editor** opens.
2	Select the top node in the tree view of the configuration. **Result**: The **Record settings** are displayed on the right-hand side of the dialog box.
3	Assign a boolean variable to the **Record Condition** field. The boolean variable controls the recording of the data.
4	Select the application that contains the trend recording and execute the command **Application > Login**.
5	Start the application.

Alarm Configuration

What's in This Chapter

General Information 481
Configuration of the Alarm Management 483
Calling Event Alarms in the Program..... 486
Alarm Management Objects..... 487

An alarm element can be added to a visualization. For details regarding the configuration of visualization alarm management, refer to *Visualizing Alarm Management* in the online help under **Software > Programming and Configuring > CODESYS Visualization > Alarm Management**.

NOTE: An alarm element can only be configured if a **Webvisualisation** object has been added to the **Visu Manager** object.

General Information

Alarm Management

Overview

You can configure an alarm management system in an application to monitor critical conditions of your machine by triggering alarms.

Alarms are defined as part of alarm groups. For each alarm, configure the following parameters:

- The monitoring criteria
- An optional user-defined deactivation variable
- An alarm class
- A message
- An optional pending time

Moreover, you can assign latch variables to lock (or mark) an alarm so that it can be filtered according to the latch values.

This function is not available for all controllers. Consult the *Programming Guide* specific to your controller.

Visualization of Elements to Program a User Interface

You can use the **Alarm Table** and **Alarm Banner** visualization elements to program a user interface for the alarm management. Alarms are visualized by the user and can be acknowledged. As an alternative, you can use the AlarmManager library to manage alarms within your application code.

For details regarding the configuration of visualization alarm management, refer to *Visualizing Alarm Management* in the **Software > Programming and Configuring > CODESYS Visualization > Alarm Management** part of the online help.

Alarm Management Objects

The following alarm management objects are available in the **Tools tree** of the Logic Builder:

- The **Alarm Configuration** object is the root object of the alarm management that is inserted below an **Application** node. By default, subnodes for **Error**, **Warning** and **Info** alarm classes and **Alarm Storage** are also created. You can insert more alarm objects according to your individual requirements.
- The **Alarm Class** object allows you to group general alarm criteria into a class. The number of **Alarm Class** objects is not restricted. The classes you define are available for selection in the **Class** column of the alarm groups editor and serve as a core configuration of an alarm.

  Examples of alarm criteria:

  - A specific acknowledgment method for this alarm class.
  - An action that is executed automatically when a specific alarm condition occurs.
  - Display options for the **Alarm Table** and the **Alarm Banner** in the visualization.
- The **Alarm Group** object allows you to create alarm definitions for individual alarms which are grouped according to a shared deactivation variable and archiving parameters. The number of **Alarm Group** objects per **Alarm Configuration** is not restricted. You can insert folder objects to create a hierarchical structure of alarm groups. For each **Alarm Group** object, a **Text List** object is created with the same name. It allows you to localize the alarm messages.
- The **Alarm Storage** object is available once per application. It allows you to configure the archiving of alarms.

For further information, refer to Configuration of the Alarm Management, page 483.

Defining Alarms

Overview

An alarm definition consists of the following components:

- General description (such as ID, message text)
- Description of the cause of the alarm (such as expression to be monitored, limits, minimum pending time)
- Description of the effects of the alarm (such as actions, display properties, confirmation method) as part of the associated alarm class

Acknowledging Alarms

The main reason for the use of alarms is to inform the user about an alarm situation. To help ensure that the user has taken the alarm information into account, Logic Builder allows you to configure different actions in the alarm class. The user must acknowledge the alarm to delete it from the alarm list. The different types of alarm acknowledgment are described in the status diagrams as status transitions, page 492.

Alarm Events - Alarm Conditions

An alarm event is linked to an alarm condition. The alarm condition can exist for a period of time, whereas the alarm event indicates a change, for example the change from the normal operating state to the alarm state.

The following names are used for the three types of events and the associated alarm states in the alarm configuration:

- **Active**
- **ACK**
- **Inactive**

Configuration of the Alarm Management

Creating an Alarm Configuration

Steps for Creating an Alarm Configuration

Step	Action
1	In the **Tools tree**, right-click the **Applications** node and execute the command **Add Object > Alarm Configuration...**. **Result**: The **Add Alarm Configuration** dialog box opens.
2	Click the **Add** button. **Result**: The alarm management is created for your application. By default, it consists of the root node **Alarm Configuration** and the subnodes for **Error**, **Warning** and **Info** alarm classes and **Alarm Storage**.

Configuring an Alarm Class

Prerequisites

The following variables must be declared in a global variable list (GVL):

```
bPartsShortage : BOOL;
iFillLevel : INT;
bPart1Empty : BOOL;
```

Steps for Configuring an Alarm Class

Step	Action
1	In the **Tools tree**, right-click the **Alarm Configuration** node and execute the command **Add Object > Alarm Class...**. **Result**: The **Add Alarm Class** dialog box opens.
2	Enter a **Name** (for example, `PartsShortage`) and click the **Add** button. **Result**: The alarm class `PartsShortage` is added as new subnode of the **Alarm Configuration** node and the `PartsShortage` alarm class editor opens on the right-hand side.
3	In the `PartsShortage` alarm class editor, configure the following parameters that belong to this class group: • Set **Priority** to **10**. • Select the **Archiving** option. • For **Acknowledgment method** select **REP_ACK**. • Select the **Acknowledge separately** option.
In the next steps, configure that the variable *bPartsShortage* gets the value TRUE when alarm messages are generated:	

Step	Action
4	In the **Notification Actions** section of the editor, double-click the **Action** column, select the option **Variable** from the list and press **Enter**. **Result**: The action is added and the options are activated.
5	Deselect the options **Confirm** and **Deactivate**. (Keep the **Activate** option selected.)
6	In the **Variable** section of the editor: • Click in the **Variable** input field and type `bPartsShortage` or use the **Input Assistant** to select the variable. • Type the value `TRUE` in the field on the right-hand side. **Result**: The statement `bPartsShortage := TRUE` is displayed in the **Details** column of the **Notification Actions** section of the editor.
7	In the **Display Options for Alarm Table / Alarm Banner** section of the editor: In the **Active** row select from the **Background Color** the entry **Red**.
8	In the **Display Options for Alarm Table / Alarm Banner** section of the editor: In the **Waiting for confirmation** row select from the **Background Color** the entry **Yellow**. **Result**: All messages in the `PartsShortage` class will now set the variable *bPartsShortage* to TRUE. Active messages are highlighted in red in the **Alarm Table** visualization element. Deactivated messages that are not acknowledged are highlighted in yellow.

Configuring an Alarm Storage

Steps for Configuring an Alarm Storage

Step	Action
1	In the **Tools tree**, right-click the **AlarmStorage** node and execute the command **Properties...**.
2	In the **Properties** dialog box, change the name from `AlarmStorage` to, for example, `DatabaseFileAlarmStorage` and click the **OK** button.
3	Double-click the node **DatabaseFileAlarmStorage** from the **Tools tree**.
4	In the **DatabaseFileAlarmStorage** alarm storage editor on the right-hand side, configure the following parameters: • The **Subdirectory** field is by default empty and the default directory */usr/Alarms* is set on the controller for the alarm SQLite files. If you enter a name, a subfolder of this name is created in the default directory. • In the **Limit** section, select the option **Maximum number of records** and enter a value, for example, `1000`.

After you have performed these alarm storage configurations, the alarms are saved internally at application runtime in an SQLite database in the directory */usr/Alarms* on the Modicon M262 Logic/Motion Controllers. The database contents can be displayed as historic alarms as part of an alarm table in a visualization. As a prerequisite, the **History** control variable of the alarm table must be set to TRUE. For further information, refer to *Visualization Element: Alarm Table* in the **Software > Programming and Configuring > CODESYS Visualization > reference: Programming > Visualization Elements** part of the online help.

Configuring an Alarm Group

Steps for Configuring an Alarm Group

After you have created an alarm class as described in Configuring an Alarm Class, page 483, create an alarm group with an alarm for monitoring the `PartsShortage`:

Step	Action
1	In the **Tools tree**, right-click the **Alarm Configuration** node and execute the command **Add Object > Alarm Group...**. **Result**: The **Add Alarm Group** dialog box opens.
2	Enter the **Name** `PartsShortage` and click the **Add** button. **Results**: • The alarm group `PartsShortage` is added as new subnode of the **Alarm Configuration** node and the `PartsShortage` alarm group editor opens on the right-hand side. • A `PartsShortage` text list is automatically created.
3	In the `PartsShortage` alarm group editor, set the parameter **Archiving** to **(none)**.
4	Double-click the **Observation Type** column and select the option **Digital** from the list.
5	Configure the following parameters for the alarm: • Set **Class** to `PartsShortage`. • For **Message** enter `Parts shortage at Station 1 – Fill level: <LATCH1>`. (`<LATCH1>` is a placeholder for the latch variable 1.) • For **Min. Pend. Time** enter `T#5s`. • For **Latch Var 1** enter `iFillLevel` or select from the **Input Assistant**.
6	In the **Digital** section of the editor: • Click in the **Expression** input field and type `bPart1Empty` or use the **Input Assistant** to select the variable. • Type the value `TRUE` in the field on the right-hand side. **Result**: The statement `bPart1Empty = TRUE` is displayed in the **Details** column of the editor.
7	From the **Archiving** list, select **DatabaseFileAlarmStorage** which you have configured with the alarm storage, page 484.

Results:

If the variable `bPart1Empty` switches to TRUE, a message is triggered. The value of the variable `iFillLevel` is issued with the message. The acknowledgment response and the display are executed according to the `PartsShortage` alarm class. The alarm will be archived.

Using Latch Variables to Extend Alarm Messages

When you define one or more alarm messages as `Message <n>` for an alarm in the alarm group editor, you can also indicate that the value of latch variables is taken over to these messages. To achieve this, insert one or more corresponding placeholders `<LATCH1>, <LATCH2>` in the message text. As a result, the alarm message will also contain the value of the latch variables when the alarm becomes active (for example, an error ID or a measurement value).

For further information, refer to *Using Latch Variables to Filter Alarm Events* in the **Software > Programming and Configuring > CODESYS Visualization > Alarm Management > Alarm Display in Online Mode** part of the online help.

NOTE: You can configure the alarm visualization so that in online mode the user can filter the alarms by the contents of a latch variable value.

Calling Event Alarms in the Program

Defining an Event Alarm in the Alarm Group

Overview

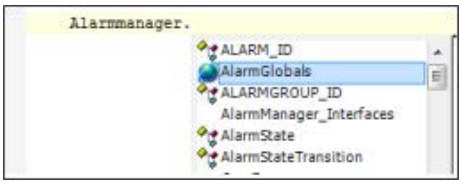
Select the **Observation Type** = **Event** to configure an **Event** alarm in an **Alarm Group**. In contrast to the other observation types which trigger the alarm depending on a condition, the **Event** alarm is triggered via a function block call from the application code.

Steps for Defining the Event Alarm in the Alarm Group

Step	Action
1	Create an **Alarm Group** named `AlarmGroup1` as described in Configuring an Alarm Group, page 485.
2	Define an alarm with **Observation Type** = **Event** in the alarm group.
3	In the column **Class**, select an alarm class with the acknowledgment method **ACK** or create a new alarm class. For details refer to Configuring an Alarm Class, page 483.

Program Call for Triggering the Event Alarm

Steps for Program Call for Triggering the Event Alarm

Step	Action
1	Place the cursor at the position in the POU where the function block for triggering the alarm is to be called.
2	Type `AlarmManager` followed by a period to use the *RaiseEvent* function from the AlarmManager library. **Result**: A smart coding window opens providing a list of elements. 
3	Select *AlarmGlobals* and enter a period. Insert the elements *g_AlarmHandler* and *RaiseEvent* accordingly. **Result**: The following line is displayed: `Alarmmanager.AlarmGlobals.g_AlarmHandler.RaiseEvent`
4	Connect the transfer variables for the parameters *AlarmGroup_ID* and *Alarm_ID* by entering a left parenthesis. **Result**: A tooltip is displayed providing information about the transfer variables.
5	Press the **F2** key. **Result**: The **Input Assistant** opens.
6	Select the **Categories** tab and select the category **Variables** from the list on the left-hand side.
7	Select the group ID variable *ID_AlarmGroup1* in this example.
8	Enter a comma and insert the alarm ID variable (*'Alm_AlarmGroup1_Alarm_IDs.ID_0'* in this example) and enter a right parenthesis and a semicolon. **Result**: The following function call is added: `Alarmmanager.AlarmGlobals.g_AlarmHandler.RaiseEvent(Alm_AlarmConfiguration_Alarmgroup_IDs.ID_AlarmGroup1, Alm_AlarmGroup1_Alarm_IDs.ID_0);`

Alarm Management Objects

Alarm Configuration

Overview

The **Alarm Configuration** object is the root object of the alarm management that is inserted below an **Application** node and cannot be configured. By default, the following subnodes are inserted:

- Alarm class **Error**
- Alarm class **Info**
- Alarm class **Warning**
- Alarm storage **Alarm Storage**

You can customize this structure by removing objects or inserting more alarm objects according to your individual requirements. For further information, refer to Configuration of the Alarm Management, page 483.

Alarm Group Template

Overview

The **Alarm group template** object is useful for library developers to define alarm conditions with variables of a custom type (function block or structure). The data type can be part of a development library. The variables of the function block can be checked for alarm conditions.

The object can be added to the **Devices tree** or to the **Global** node of the **Tools tree**. A text list object with the same name is automatically created.

Alarm Group Template Editor

Double click the **Alarm group template** node to open the **Alarm group template** editor. It provides the following elements:

Element	Description
Text list	Name of the text list object that is created with the alarm group template object. Default name: **AlarmGroupTemplate** By default, the automatically generated text list is selected. You can select another text list from the list.
Deactivation	Boolean variable to control the deactivation of the alarms in this group. **NOTE:** Alarms that have been deactivated are not reported, even if the alarm condition is fulfilled.
Type for alarm definition	Interface variable for alarm definitions.
inst : <type>	Type (function block or structure) of the alarm instance. The interface variable allows access to the variables of the customized type. Alarm conditions can be defined.

Table of Alarms

Below the configuration elements, the **Alarm group template** editor contains the table of alarms. Enter the properties of the individual alarms in the corresponding table columns. Depending on the selected **Observation Type**, input fields are provided below the table for additionally required parameters. For details, refer to Additional **Observation Type** Parameters Below the Alarm Table, page 489.

Table Column	Description
ID	Alarm ID that is automatically assigned. The ID is a combination of the instance name and the ID from the library. Example: *GVL.g_modul1Fan1* Function block *FAN_ALARM_PRIO* with the resulting alarm ID *GVL_g_modul1Fan1_ID_FAN_ALARM_PRIO*.
Observation Type	Observation type selected from the list. Depending on the selected **Observation Type**, input fields are provided below the table for additionally required parameters. For details, refer to Additional **Observation Type** Parameters Below the Alarm Table, page 489. The fields describe an expression for triggering the alarm.
Details	Displays the configuration settings that are defined in the fields below the table for the selected alarm type. Alternatively, you can enter the condition directly in the field. If you enter an expression that does not match the selected **Observation Type**, the type is automatically adapted.
Deactivation	You can optionally configure a variable for deactivating the alarm.
Class	Select an alarm class from the list. Example: *AGL.ACL.Error*

Message	Enter text for a message that is displayed in the **Alarm Table** visualization element when the alarm is detected. The text is managed in the text list of the alarm group.
	
	You can extend the message with information from latch variables. To insert line breaks, press **Ctrl** + **Enter**.
	
	The following placeholders are supported.
	• `<DATE>`: Date when the change to the present state was triggered.
	• `<TIME>`: Time when the change to the present state was triggered.
	• `<EXPRESSION>`: Expression (defined within the alarm) that triggered the alarm.
	• `<PRIORITY>`: Priority of the alarm (configured in the alarm class editor, page 490).
	• `<TRIGGERVALUE>`*: Value that caused the alarm condition to become TRUE.
	• `<ALARMID>`: Alarm ID as displayed in the first column of the alarm table.
	• `<CLASS>`: Name of the alarm class (refer to Configuring an Alarm Class, page 483).
	• `<ALLDEFAULT>`: Complete information about the alarm.
	• `<CURRENTVALUE>`*: Present value of the monitored variable.
	• `<LATCH1>`*: Value of the first latch variable at the time when the alarm was triggered. To use more than the two default latch variables, insert additional columns from the contextual menu.
	• `<LATCH2>`*: Value of the second latch variable at the time when the alarm was triggered. To use more than the two default latch variables, insert additional columns from the contextual menu.
	• `<ALARM>`: TRUE, if the alarm state is *Active*. FALSE for other alarm states.
	• `<STATE>`: Alarm state:
	  ◦ 0 = *normal*
	  ◦ 1 = *active*
	  ◦ 2 = *waiting for ACK*
	  ◦ 3 = *active, acknowledged*
	***** For `<TRIGGERVALUE>`, `<CURRENTVALUE>` and `<LATCHn>`, you can additionally use formatting specifications such as those that are valid for the C-function `printf`. Example: `The value is <CURRENTVALUE %d>`
	
	By default, one column is available for message text. You can add more columns from the contextual menu that is available in the **Data type** property for column configuration of the **Alarm table** and **Alarm banner** visualization elements if configured in the **Tools > Customize** dialog box.

On-Delay time	Time for switch-on delay. Defines the minimum time that the alarm condition must be fulfilled before the alarm is triggered. With this parameter, you can specify that alarm states that are TRUE for less than the configured time will not trigger an alarm. The parameter must have an IEC 61131-compliant format. Example: `T#2s`

Table Column	Description
Off-Delay time	Time for switch-off delay. Defines the minimum time that the alarm condition must be inactive before an alarm is treated as inactive. Similar to the **On-Delay time**, this parameter can be used to ignore short-term value changes. Example: `T#4s`
Latch Var 1	Reserved for instance name. The instance name is generated as a STRING in *LatchVar1* so that the alarms can be assigned to an instance.
Latch Var 2	Another latch variable in order to record additional values when the alarm becomes active. Data type: Scalar (8 bytes) or of type STRING or WSTRING The latch variable or the latch expression is recorded when the alarm is triggered and provides additional information about the alarm. By default, columns are available for two latch variables. You can add more columns from the contextual menu that is available in the **Data type** property for column configuration of the **Alarm table** and **Alarm banner** visualization elements if configured in the **Tools > Customize** dialog box.
Higher Prio. Alarm	When the alarm is triggered, it can be acknowledged by triggering another alarm. Example: A temperature monitor is configured to issue a warning message at temperatures > 60° C and an error messages at temperatures > 90 °C. The warning message is displayed when the temperature reaches 60° C. When the temperature reaches 90° C, the 60° C warning message is acknowledged and the 90 °C error message is displayed.

Additional Observation Type Parameters Below the Alarm Table

Depending on the selected **Observation Type**, below the table of alarms, input fields are provided for additionally required parameters.

Observation Type	Additional parameters
Digital	**Expression**: Enter the expression to be monitored or select it from the **Input Assistant**. On the right-hand side, enter the expression for the comparison. In between both expressions, select the comparison operator from the list (= or <>).
Upper limit[1)]	• **Expression**: As described for **Digital**, but with the comparison operators > or >= . • **Hysteresis in %** (optional)
Lower limit[1)]	• **Expression**: As described for **Digital**, but with the comparison operators < or <= . • **Hysteresis in %** (optional)
Outside range[1)]	The alarm is triggered when the monitored expression is outside the defined range. • **Expression** to be monitored. • **Area**: On the left-hand side, enter the expression for the lower limit of the range. On the right-hand side, enter the expression for the upper limit. The expression to be monitored is displayed in the middle. Set the comparison operators in-between the three fields accordingly. • **Hysteresis in %** (optional)
Inside the range[1)]	The alarm is triggered when the monitored expression is inside the defined range. • **Expression** to be monitored. • **Area**: On the left-hand side, enter the expression for the lower limit of the range. On the right-hand side, enter the expression for the upper limit. The expression to be monitored is displayed in the middle. Set the comparison operators in-between the three fields accordingly. • **Hysteresis in %** (optional)
Change	**Expression**: Enter the expression to be monitored or select it from the **Input Assistant**. An alarm is triggered when the value changes.
Event	An event alarm is triggered by using functions from the AlarmManager library.

[1)] For these observation types, you can monitor the specified expression as absolute or relative. Absolute means that the limit value is defined by a fixed value or a variable that returns a fixed value. Relative means that the limit value depends on an expression; for example: upper limit: `Variable x >= 0.9 * y`.

Hysteresis in %: When you configure a hysteresis, the alarm condition is fulfilled until a certain deviation from the specified limit value is reached. The size of the deviation is specified as a percentage [%] of the limit value.

Example for **Upper limit**: `i_temp >= 30 Hysteresis: 10%`. The alarm is active as soon as the variable *i_temp* reaches or exceeds the value 30. The alarm is no longer active if the value falls below 27.

Commands of the Contextual Menu

The contextual menu includes the following commands for adding and removing additional columns for messages and latch variables:

- **Add message column**
- **Remove message column**
- **Add latch var column**
- **Remove latch var column**

Select multiple rows in the table to apply the commands of the contextual menu simultaneously.

Exporting and Importing Alarms

The contextual menu of the alarm tables provides the following commands for the export and import of alarms to and from a CSV file:

- **Export Alarms**: The command opens a default **Save As** dialog box. It allows you to export the open alarm list to a file of type **Alarm Group Export File (*.csv)**.
- **Import Alarms**: The command opens a default dialog box for opening a file of type **Alarm Group Export File (*.csv)** that describes alarm definitions.

Alarm Class

Overview

An alarm class describes the general properties of an alarm. The configured alarm classes are available for selection in the **Class** column of the editor of the alarm group. They serve as a core configuration of an alarm and define the following:

- Specific acknowledgment method (how the user must confirm the alarm).
- Action execution (what happens automatically if an alarm conditions occurs).
- Display options for the **Alarm Table** or the **Alarm Banner** (how the alarms are visualized).

For further information, refer to Configuration of the Alarm Management, page 483.

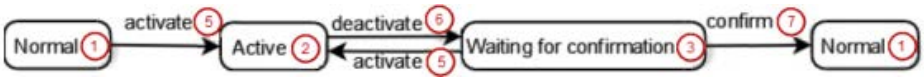
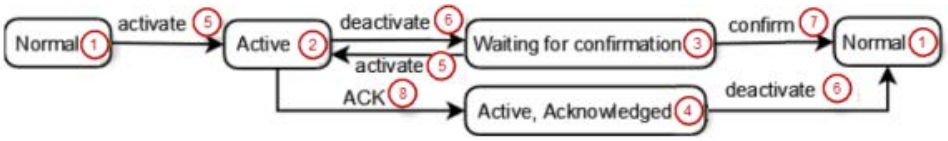
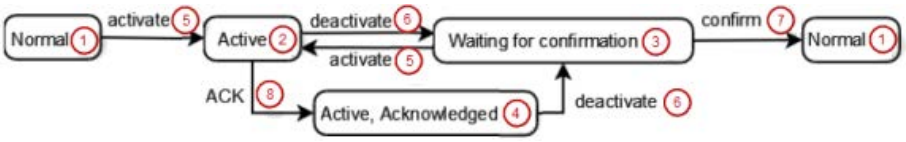
Elements of the Alarm Class Editor — General Section

Element	Description
Priority	Specifies the priority of the alarm condition in this class (0...255). 0 represents the highest priority, 255 the lowest priority.
Archiving	If selected, the alarms of this class are recorded. Create an **Alarm Storage** node to configure the archiving parameters.
Acknowledgment section	

Element	Description
Acknowledgment method	Defines how an alarm is acknowledged: • **REP**: Alarm is set to inactive after the cause has been resolved. • **ACK**: Alarm is set to inactive after it has been acknowledged; acknowledgment method for **Event** alarms. • **REP_ACK**: Alarm is set to inactive after the cause has been resolved and the alarm has been acknowledged. • **ACK_REP**: Alarm is set to inactive after the alarm has been acknowledged and the cause has been resolved. • **ACK_REP_ACK**: Alarm is set to inactive after optional acknowledgment that the alarm was received, the cause has been resolved and acknowledgment that the alarm situation was ended. **NOTE:** You have to ensure that the user notices the alarm during operation. To achieve this, use the actions that are available in the **Alarm Class**. The user must confirm (acknowledge) the alarm in order to remove the alarm from the alarm list. Also refer to Acknowledgment Methods — Detailed Information, page 492.
Acknowledge separately	If selected, the user cannot acknowledge the alarm together with other alarms during operation but must acknowledge it separately. This helps to prevent the user from inadvertently acknowledging an alarm with other alarms. **NOTE:** If this option is selected, the **Acknowledge all visible** button of the **Alarm Table** element in the visualization does not acknowledge all alarms.

Acknowledgment Methods — Detailed Information

The table provides the acknowledgment methods as diagrams depending on the alarm states and status transitions:

Element	Description
REP	
ACK	
REP_ACK	
ACK_REP	
ACK_REP_ACK	

Alarm status:

- **1 - Normal** No alarm situation.
- **2 - Active** An alarm situation has been detected.
- **3 - Waiting for confirmation** The alarm situation no longer applies, an acknowledgment has not been performed.
- **4 - Active, Acknowledged** The alarm situation still applies, the alarm has been acknowledged.

Status transitions:

- **5 - Activate** An alarm situation is detected (“alarm coming”).
- **6 - Deactivate** The alarm situation no longer applies (“alarm going”).
- **7 - Confirm** Acknowledgment of an alarm that no longer applies.
- **8 - ACK** Acknowledgment of an alarm that still applies.

NOTE: The status transition **Confirm** and **ACK** can be also managed by the **Acknowledge selected** and **Acknowledge all visible** buttons of the **Alarm Table** element in the visualization. However, with the **ACK** status transition the alarm is still active.

For details, refer to:

- *Control variables* in the **Software > Programming and Configuring > CODESYS Visualization > reference: Programming > Visualization Elements > Visualization Element: Alarm Table > Element properties** part of the online help
- *Visualizing Alarm Management* in the **Software > Programming and Configuring > CODESYS Visualization > Alarm Management** part of the online help

Elements of the Alarm Class Editor — Notifications Actions Section

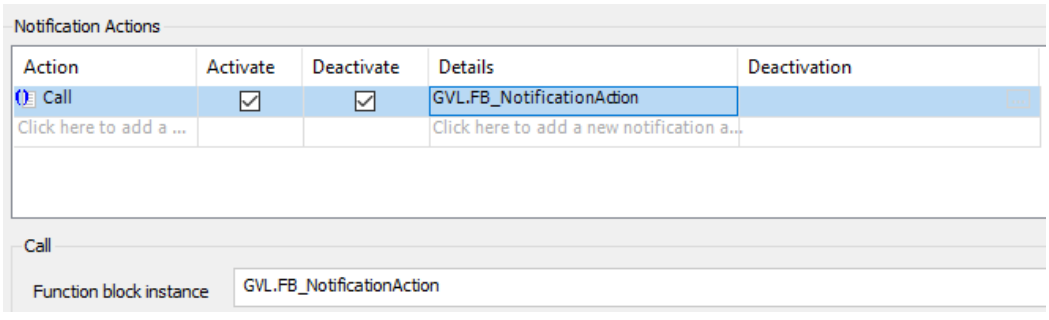
You can assign a list of actions to each alarm class to be executed as soon as the alarm is subject to a status transition.

Element	Description
Action	Click the **Action** column and select one of the following actions: • **Variable**: A value is assigned to a variable. • **Execute**: A program is executed. • **Call**: A function block instance is called.
Activate **Deactivate** **Confirm** **Details**	If selected, the action is executed with the status transition. The available status transitions depend on the **Acknowledgment method** that has been selected.
Details	Displays a summary of the action. Define the action in the section below the table (the **Details** section of the selected **Variable**, **Execute** or **Call**) or edit the action directly in this row.
Deactivation	Defines a variable that allows an application to deactivate the action.

Elements of the Alarm Class Editor — Variable, Execute or Call Section

The **Details** section changes to **Variable**, **Execute** or **Call** depending on the **Action** selected in the **Notifications Actions** section.

Action	Elements
Variable	Contains the variable to which you assign a value or an expression on the left-hand side and the assigned variable or expression on the right-hand side. You can use the following placeholders: • For Boolean values use `ALARM`: Status (TRUE or FALSE) of the alarm of this alarm class that was last activated/deactivated. • For integer variables use `STATE`: Status of the alarm of this alarm class that last changed its status. (0: *Normal*, 2: *Active*, 3: *WaitingForConfirmation*, 4: *ActiveAcknowledged*) For string variables you can enter any IEC literal (for example, `'too high!'`).
Execute	The **Action > Execute** is unsupported.

Call	• **Function block instance**: Enter the function block that is called. The function block must implement the interface `IAlarmNotifiable` (`AlarmManager.library`). The action to be executed on a change of state is implemented in the method *Execute*.
	• **Additional parameter structure**: If the function block requires an additional parameter structure, this table lists the members of the structure and allows you to enter data. An additional parameter structure is defined in the function block with the attribute *AlarmManagerAdditionalData*.
	Examples:
	
	The structure must consist of scalar components or components of the type `POINTER TO`. User-defined data types or ARRAY data types are unsupported.
	
	NOTE: You can mark structure members as mandatory parameters by using the attribute `{attribute 'AlarmManagerMandatoryParameter'}`.
	
	NOTE: The function blocks to be called need to be instantiated to be accessible from the **Notification Actions**.
	
	`VAR_GLOBAL`
	`{attribute 'AlarmManagerAdditionalData' := 'StructAlarmParams'}`
	`FUNCTION_BLOCK FB_IAlarmNotifiable_Implemented IMPLEMENTS IAlarmNotifiable`
	`VAR`
	`iCounter: INT;`
	
	`pAlarmParams: POINTER TO StructAlarmParams;`
	`sParamString: STRING;`
	`iParamNum: INT;`
	
	`psMessage: POINTER TO STRING;`
	`sMessage: STRING;`
	`END_VAR`
	
	`TYPE StructAlarmParams :`
	`STRUCT`
	`str: STRING;`
	`{attribute 'AlarmManagerMandatoryParameter'}`
	`num: INT;`
	`END_STRUCT`
	`END_TYPE`
	
	`pAlarmParams := pbyAdditionalData;`
	`sParamString := pAlarmParams^.str;`
	`iParamNum := pAlarmParams^.num;`
	
	`iCounter := iCounter + 1;`
	
	`psMessage := itfAlarm.GetMessage(stringType := __SYSTEM.TYPE_CLASS.TYPE_STRING);`
	`sMessage := psMessage^;`
	`FB_NotificationAction: FB_IAlarmNotifiable_Implemented ;`
	`END_VAR`
	
	

Elements of the Alarm Class Editor — Display Options for Alarm Table / Alarm Banner Section

Defines how the individual alarm state is displayed in the alarm visualization element **Alarm Table** or **Alarm Banner**.

Element	Description
State	The available alarm states depend on the **Acknowledgment method** that has been selected.
Font	Click the browse **...** button to open the default dialog box for selecting the text font and color.
Background Color	Select the background color for the alarm message from the list.
Bitmap	You can add a bitmap by entering the bitmap ID. If no image file is assigned to this ID yet, the default file open dialog box is displayed. After you have selected a file, the bitmap is added to the **GlobalImagePool**.
Transparent	Activate this option if the bitmap has a transparent color.
Transparent color	This parameter only applies if the option **Transparent** is selected: Select the color that shall be displayed transparently.

Alarm Group

Overview

Use alarm groups for organizing individual alarms in the alarm management of an application. Assign each alarm to only one alarm group. To configure an alarm group, insert an **Alarm Group** object below an **Alarm Configuration** object in the **Tools tree**. The alarm group editor allows you to define settings for an entire group of alarms (such as the archiving type, a shared deactivation variable, and a shared text list). You can create a hierarchical structure of alarm groups in the **Tools tree** by using folder elements.

The alarm group editor provides a table listing the alarms of the group and the properties of each alarm. You can edit the values in the list. An export and import function to and from a CSV file is also available, page 498.

For further information, refer to Configuration of the Alarm Management, page 483.

Elements of the Alarm Group Editor - General Section

Element	Description
Text list	Name of the text list where the alarm message text is saved (also refer to the parameter **Message** in the next paragraph). It is created automatically when an alarm group is added. The text list can be used for the localization of the messages. You can also select another text list from the list.
Archiving	If you select an alarm storage object from this list, alarm storage is automatically activated.
Deactivation	Enter or select a Boolean variable that deactivates all alarms in this group. When this variable is applied, the alarms will no longer be triggered even if the alarm condition is fulfilled. Alarms that are already active are not affected.

Elements of the Alarm Group Editor - Table of Alarms

Enter the properties of the individual alarms in the table columns. Depending on the selected **Observation Type**, additional parameters are provided in the input fields below the table.

Column	Description
ID	This unique ID corresponds to the ID in the associated text list. You can modify the ID in the **ID** field of the table. The ID must be unique within the alarm group. A change of the ID in the alarm table directly changes the ID in the text list and vice versa. The alarm verification is performed in ascending order, starting with **ID** = 0. It is a good practice to list the alarms in the order of priority to make them visible/hidden when multiple conditions are fulfilled. To raise the alarm with the highest priority if multiple alarms are triggered, it must be assigned a smaller ID value than the alarm that is displayed in the **Higher Prio. Alarm** column.
Observation Type	Select an observation type from the list. Depending on the type, different editing fields are available below the table that define the condition for the alarm. The fields define an expression that triggers the alarm. Refer to the table describing the observation types, page 496.
Details	Displays the configuration settings for the selected alarm type. Define the settings in the section below the table for the selected **Observation Type**: **Digital**, **Upper limit**, **Lower limit**, **Outside range**, **Inside the range**, **Change** or **Event** or edit the action directly in this **Details** field. If you enter an expression that does not match the selected **Observation Type**, the expression is automatically adapted.
Deactivation	Enter an optional Boolean variable for deactivating the alarm.
Class	Select an alarm class from the list.
Message	Enter text for the alarm message that is displayed in an **Alarm Table** visualization element when the alarm occurs. This text is automatically transferred to the text list associated with the alarm group. You can insert line breaks by pressing **Ctrl**+**Enter**. For a list of supported placeholders that are replaced by the present value at runtime, refer to the separate table in this chapter, page 497. **NOTE:** By default, only one column is available for message text. You can add more columns from the contextual menu that is available in the **Type of data** property for **Column** configuration of the **Alarm table** and **Alarm banner** visualization elements.
Min. Pend. Time	Enter the minimum time the alarm condition must apply before the alarm is triggered. This allows you to inhibit alarms for conditions that occur only for a short time. Use an IEC 61131-compliant format, for example, `t#2ms`.
Latch Var 1 **Latch Var 2**	The latch variables are used to record additional values when the alarm becomes active. A latch variable must be a scalar type (8-byte) or a STRING or WSTRING type. A latch variable or a latch expression is recorded when the alarm becomes active. This allows you to provide additional information about the alarm. **NOTE:** The **Alarm table** or **Alarm banner** elements allow you to filter the alarms in the visualization by the contents of the latch variable. See how to include latch variables in the alarm **Message** in the list of supported placeholders, page 497. Example: When monitoring a motor speed, the temperature and the current are additionally recorded. By default, columns are available for two latch variables. You can add more columns from the contextual menu that is available in the **Data type** property for column configuration of the **Alarm table** and **Alarm banner** visualization elements if configured in the **Tools > Customize** dialog box.
Higher Prio. Alarm	When the alarm is triggered, it can be acknowledged by triggering another alarm. Example: When monitoring a temperature, an alarm of class **Warning** should be triggered at a temperature > 60 °C and an alarm of type **Error** at a temperature > 90 °C. If the temperature rises, the alarm of class **Warning** is displayed starting at 60 °C. At 90 °C and higher, the alarm of type **Error** is triggered and automatically acknowledges the 60 °C alarm.

Observation Type Values

The **Details** section changes to **Digital**, **Upper limit**, **Lower limit**, **Outside range**, **Inside the range**, **Change** or **Event** depending on the **Observation Type** selected.

Observation Type	Elements
Digital	Enter the **Expression** to be monitored on the left-hand side and the expression for the comparison on the right-hand side. In the middle, select the comparison operator **=** or **<>**.
Upper limit(1)	Enter the (left and right) **Expression** as described for the **Digital** type and select the comparison operator **>** or **>=**. Optionally, enter a **Hysteresis in %**.
Lower limit(1)	Enter the (left and right) **Expression** as described for the **Digital** type and select the comparison operator **<** or **<=**. Optionally, enter a **Hysteresis in %**.
Outside range(1)	Enter the **Expression** to be monitored. **Area**: The alarm is triggered when the monitored expression is outside the range defined here. The left field defines an expression for the lower limit of this range. The right field defines an expression for the upper limit. The expression to be monitored is displayed in the field in the middle. Set the comparison operators **<** or **<=**accordingly. Optionally, enter a **Hysteresis in %**.
Inside the range(1)	Enter the **Expression** to be monitored. **Area**: The alarm is triggered when the monitored expression is within the range defined here. The left field defines an expression for the lower limit of this range. The right field defines an expression for the upper limit. The expression to be monitored is displayed in the field in the middle. Set the comparison operators **<** or **<=**accordingly. Optionally, enter a **Hysteresis in %**.
Change	Enter the **Expression** to be monitored. An alarm is triggered when the value changes.
Event	The **Event** alarm is triggered by a function block call from the library AlarmManager.library. For further information, refer to Calling Event Alarms in the Program, page 486.
(1) For these **Observation Types**, you can monitor the specified expression as absolute or relative. Absolute means that the limit value is defined by a fixed value or a variable that returns a fixed value. Relative means that the limit value depends on an expression. Example: Upper limit: `Variable x >= 0.9 * y`. **Hysteresis in %**: When you specify a hysteresis, the alarm condition is fulfilled until a certain deviation from the specified limit value is reached. The size of the deviation is specified as a percentage [%] of the limit value. Example: Upper limit: `i_temp >= 30 Hysteresis: 10%` The alarm is active when the variable *i_temp* reaches or exceeds the value 30. The alarm becomes inactive when the value falls below 27.	

Placeholders Supported for Alarm Messages

The following placeholders are supported for alarm messages and will be replaced by the present value at runtime.

Placeholder	Description
`<DATE>`	Date of the change to the present status.
`<TIME>`	Time of the last status change.
`<EXPRESSION>`	Expression (defined within the alarm) that triggered the alarm.
`<PRIORITY>`	Priority of the alarm (defined in the alarm class).
`<TRIGGERVALUE>`(1)	Value that caused the alarm condition to become true.
`<ALARMID>`	Alarm ID as displayed in the first column of an alarm table.
`<CLASS>`	Name of the alarm class (defined in the alarm).
`<ALLDEFAULT>`	The complete information about the alarm is displayed.
`<CURRENTVALUE>`(1)	Present value of the monitored variable.
`<LATCH1>`(1)	Value of the first latch variable at the time when the alarm was triggered.
`<LATCH2>`(1)	Value of the second latch variable at the time when the alarm was triggered.
`<ALARM>`	TRUE in the case of the alarm status *Active*. FALSE with every other alarm status.

Placeholder	Description
<STATE>	Alarm status: 0 = *Normal*, 1 = *Active*, 2 = *WaitingForConfirmation*, 3 = *ActiveAcknowledged* (still active and already acknowledged)
(1) For <TRIGGERVALUE>, <CURRENTVALUE> and <LATCHn> you can also use formatting specifications, such as those that are valid for the C function printf (for example: The value is <CURRENTVALUE %d>).	

Commands of the Contextual Menu

You can make the following commands for adding and removing additional columns for messages and latch variables available in the contextual menu with the **Tools > Customize** command:

- **Add message column**
- **Remove message column**
- **Add latch var column**
- **Remove latch var column**

Export / Import of Alarms

You can make the following commands for exporting and importing alarms to and from a CSV file available in the contextual menu with the **Tools > Customize** command:

- **Export Alarms**

  The command opens the default dialog box for saving a file in the file system. Select the default file type **Alarm Group Export File (*.csv)** and click **OK** to export the present alarm list to a CSV file.
- **Import Alarms**

  The command opens the default dialog box for searching for a file in the file system. Select a file of type **Alarm Group Export File (*.csv)** that describes alarm definitions and click **OK** to import the alarms.

Alarm Storage

Overview

The **Alarm Storage** object allows you to configure the settings for the database file where the detected alarms are saved. You can insert the object only once below an **Alarm Configuration** node.

There is exactly one storage file located in a database. The default directory for the alarm SQLite files is: */usr/Alarms*. You cannot change the name of this file as it is derived from the application name according to the following convention: <application name>.alarmstorage.sqlite. The alarm classes and alarm groups define whether or not the storage file should be used.

For further information, refer to Configuration of the Alarm Management, page 483.

NOTICE

LOSS OF DATA

Backup the alarm storage file before executing the **Reset origin** command as this command clears the contents of the file.

Failure to follow these instructions can result in equipment damage.

Elements of the Alarm Storage Editor

Element	Description
Subdirectory	Optionally, enter a name of a subdirectory on the controller where the storage file can be stored.
No limit	Unsupported option: A build error is generated if this option is selected.
Maximum number of records	When selected, enter a number of recorded entries that are stored. When this number is reached and a new entry is queued in the circular buffer, the oldest entry is deleted. Value range: 1147...10000 for Modicon M262 Logic/Motion Controllers
Maximum storage size	When selected, enter the maximum size of the storage file and select the corresponding unit (**B**...**MB)** from the list. Internally the size is implicitly converted into an approximate maximum number of entries that can be saved in the file (circular buffer). Value range: 224 KB...1 MB for Modicon M262 Logic/Motion Controllers
Circular buffer	This option is unavailable as it is selected by the **No limit** option which is not supported.

The records that are stored in the database can be displayed in the **Alarm Table** visualization element.

Command CSV Export of Alarm Storage

If the application is in online mode and an alarm storage is configured, execute this command via the contextual menu of the **Alarm Storage** node to open the **Alarm Storage Export Configuration** dialog box.

Elements of the Alarm Storage Export Configuration Dialog Box - Basic Settings

In the **Export file** text box enter a path to the generated CSV file in the file system.

Element	Description
Alarms to Export	
Export all	All listed alarms are exported.
Export alarms of timestamp range (timestamp activate)	Define a time range for the export of the listed alarms. This refers to the time stamp when the alarm was activated. The **From** and **To** parameters indicate the minimum/maximum time stamps that are available in the alarm storage or that were the last time stamps that were used in an export. When selecting the time range, the smallest unit is one second.
Language	Select a language from the list. It provides all languages that are available in text lists for alarm messages in addition to the default language **(default)**.

Elements of the Alarm Storage Export Configuration - Advanced Settings Dialog Box

Element	Description
Sorting by Timestamp Activate	
Ascending Descending	Select the sort order by the time stamp when the alarm was activated.
File Format	
Formatted values and timestamps	When selected, the time stamps are formatted with the date and time format as defined below. REAL and LREAL values are formatted in numerical format.
Raw data of alarm storage file	When selected, the time stamps and values are written as unformatted text to the CSV file as they appear in the alarm storage. Alarms, alarm groups, and alarm classes are written to the CSV file as internal IDs.

Element	Description
Formatting Options Defines the formatting of the entries in the CSV file. The variables are only available if the option **Formatted values and timestamps** is selected.	
REAL format (printf syntax)	Default value: **%.1f** Use possible formats according to the printf syntax for floating point values.
Date format	When selected, the settings from the control panel (regional settings) are applied.
Time format	When selected, the settings from the control panel (regional settings) are applied.
Separator Character	
Semicolon	When selected, the entries in the CSV file are separated by a semicolon.
Comma	When selected, the entries in the CSV file are separated by a comma.
TAB	When selected, the entries in the CSV file are separated by a tab character.
Local time	This option is only available if the option **Formatted values and timestamps** is selected. When selected, the time stamps are written to the CSV file using local time. When not selected, the time stamps are written to the CSV file using universal time coordinated (UTC).

Unit Conversion

What's in This Chapter

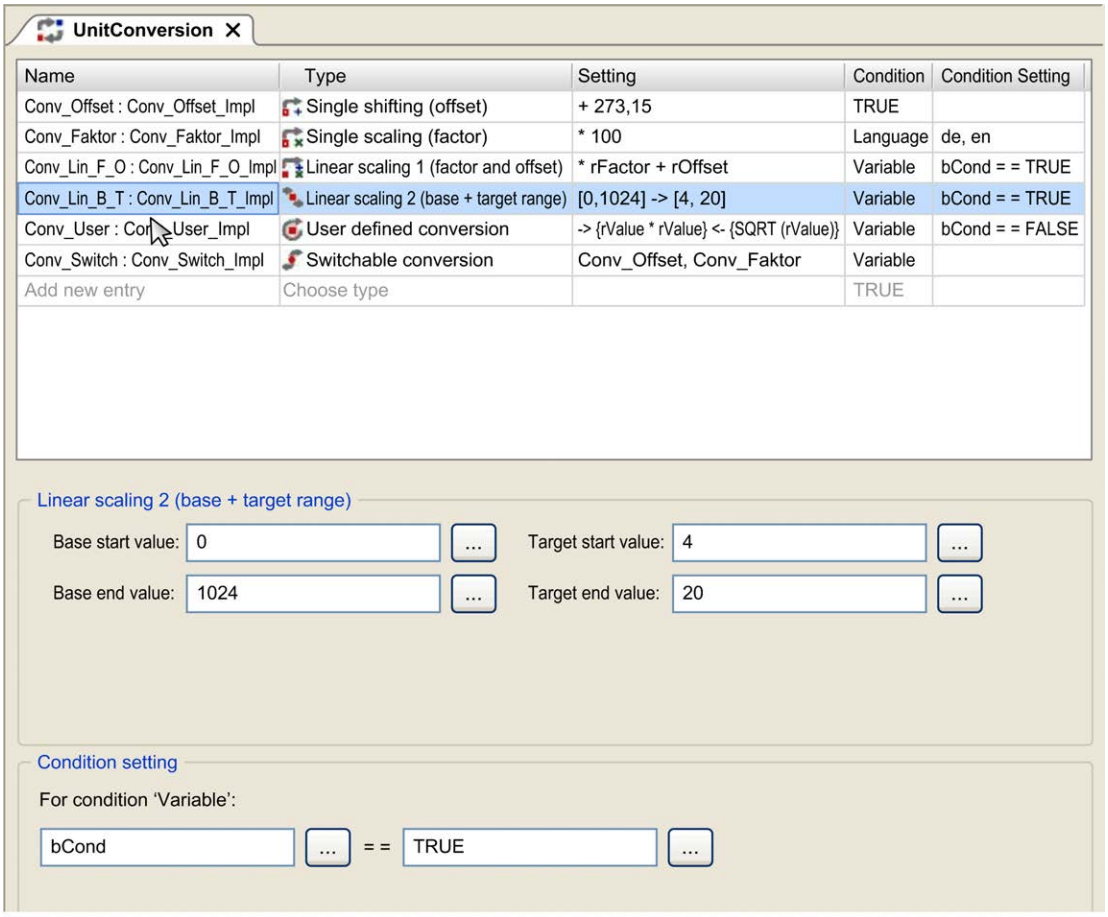
Unit Conversion Configuration 501
Usage in IEC Editors 503

Unit Conversion Configuration

Overview

In the configuration editor **UnitConversion**, you define conversions which can be used for variables in visualization elements and in the IEC editors. Add the editor as an object **UnitConversion** in the **Applications tree**. You can rename it there.

UnitConversion editor



You can define various conversions for your project. Enter a **Name** and choose the **Type** for each. The respective conversion formula is automatically displayed in the **Setting** column. Enter a **Condition** to determine when the conversion should be executed. Depending on the selected **Type**, define specific parameters in the area below the table.

The editor provides the following conversion types:

- Calculation with an offset: **Single shifting (offset)**, page 502
- Calculation with a factor: **Single scaling (factor)**, page 502
- Calculation with a factor and an offset: **Linear scaling 1 (factor and offset)**, page 502
- Calculation with the base range and target range: **Linear scaling 2 (base and target range)**, page 503

- Calculation with a user-defined expression: **User-defined conversion**, page 503
- Different calculations depending on a variable value: **Switchable conversion**, page 503

For usage of unit conversions in IEC editors, refer to the chapter *Usage in IEC Editors*, page 503.

For usage of unit conversions in visualizations, refer to the chapter *Using Unit Conversion* in the *Visualization* part of the online help.

Conditions

You can select 3 conditions to define when the calculation is executed:

Condition	Description
TRUE	The conversion is always performed.
Language	Language of the visualization (value of variable *VisuElems.CurrentLanguage*)
Variable	The conversion is performed depending on a variable value. The compare value can be a constant, a variable or an IEC expression.

Single Shifting (Offset)

Use this conversion if you want to add an offset to the input value.

(output value = input value + offset)

Parameter	Description
Offset	Value or variable

Single Scaling (Factor)

Use this conversion if you want to multiply the input value by a factor.

(output value = input value * factor)

Parameter	Description
Factor	Value or variable

Linear Scaling 1 (Factor and Offset)

Use this conversion if you want to multiply the input value by a factor and to add an offset.

(output value = (input value * factor) + offset)

Parameter	Description
Factor	Value or variable
Offset	Value or variable

Linear Scaling 2 (Base and Target Range)

Use this conversion if you want to calculate the value by defining the input and output range. A factor and an offset are calculated internally.

Parameter	Description
Base start value	Lower value of the input range
Base end value	Upper value of the input range
Target start value	Lower value of the output range
Target end value	Upper value of the output range

Example

```
Base start value: 0
Base end value: 1024
Target start value: 4.0
Target end value: 20.0
```

User-Defined Conversion

Use this conversion if you want to calculate the input value by a user-defined expression. Use the variable *rValue* as input value. In this expression, all IEC operands can be used.

Parameter	Description
Convert	Expression to convert the input value
Reverse	Expression to revert the calculated value

Switchable Conversion

Use this conversion if you want to use different conversions, depending on the language or a variable value. All defined conversions can be used.

Usage in IEC Editors

Overview

You can use unit conversions in all IEC editors. If you create a conversion for each new conversion, a function block is created automatically.

The function block includes 2 methods:

Method	Description
`Convert`	Calculation of the input value according to the conversion formula.
`Reverse`	Reverse calculation back to the input value.

You can call these methods in all IEC editors.

Example in ST

In this example, *Conv_4_20* is the name of the conversion.

```
VAR
    rValue : REAL;
    rConvertedValue : REAL;
END_VAR
rConvertedValue := Conv_4_20.convert(rValue);
```

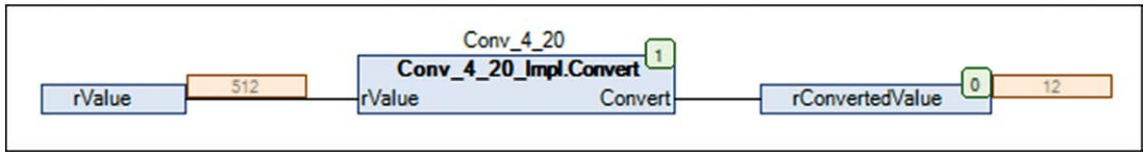
Reverse calculation back to the input value.

```
rValue := Conv_4_20.reverse(rConvertedValue);
```

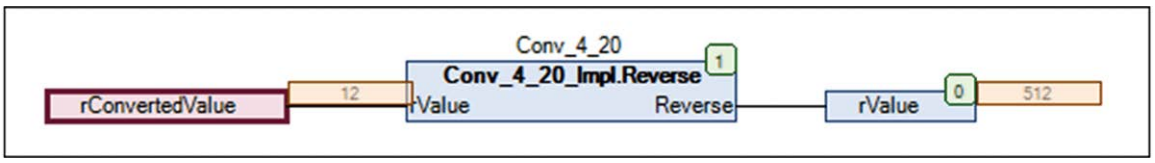
Example in CFC

In the CFC editor, enter the instance name of the conversion above the element. In the element, the method is defined.

Example in CFC method `Convert`:



Example in CFC method `Reverse`:



Symbol Configuration Editor

What's in This Chapter

Symbol Configuration Editor 505
Symbol Configuration 511
Adding a Symbol Configuration 512

Symbol Configuration Editor

Overview

The symbol configuration functionality allows you to create symbol descriptions. The symbols and the variables they represent can then be accessed by external applications, such as Vijeo Designer, EcoStruxure Operator Terminal Expert, or OPC server.

To configure symbols for an application, double-click the **Symbol Configuration** node in the **Tools tree**. The **Symbol configuration** editor view opens.

The editor contains a table. Depending on the set filter, it shows the available variables, or just those already selected for the symbol configuration. For this purpose, the concerned POUs or libraries are listed in the **Symbols** column. You can expand them in order to show the particular variables.

Elements of the Toolbar

Element	Description	
View button	The **View** button allows you to set the following filters to reduce the number of displayed variables:	
	Unconfigured from Project	Even variables not yet added to the symbol configuration, but available for this purpose in the project, are displayed.
	Unconfigured from Libraries	Also variables from libraries, not yet added to the symbol configuration, but available for this purpose in the project, are displayed.
	Symbols exported via attribute	This setting is effective only when the unconfigured variables are displayed (see the two filters described above). It has the effect that also those variables are listed, which are already selected for getting symbols by {`attribute 'symbol' := 'read'`} within their declaration. Such symbols are displayed grayed. The **Attribute** column shows which access right is set for the variable by the pragma. Refer to the following description of the **Access Rights** column, page 508.
Build button	The **Build** button allows you to build the project. The build run refreshes the variables view in the configuration editor.	
Settings button	The **Settings** button allows you to activate the following options:	
	Support OPC UA features	Select this option to download the following additional information from the OPC UA server to the controller when downloading the symbol configuration: • Base types of inherited function blocks • Contents of attributes that were assigned with compiler pragmas • Scopes (such as `VAR_INPUT`, `VAR_OUTPUT`, `VAR_IN_OUT`)
	Include comments in XML	This has the effect that comments assigned to variables are also exported to the symbol file.
	Include node flags in XML	This has the effect that the flags that contain the namespace are also exported to the symbol file. They provide additional information on the origin of the node in the namespace when OPC UA is active.
	Configure comments and attributes...	Opens the **Comments and Attributes** dialog box that allows you to configure the contents of the symbol configuration and the XML file.
	Configure synchronisation with IEC tasks...	Opens the **Options** tab of the **Properties** dialog box of the selected controller. Refer to the description of the **Access variables in sync with IEC tasks** option in the Menu Commands Online Help (see Menu Commands, Online Help). **NOTE:** Do not activate the option **Configure synchronisation with IEC tasks...** for motion and real time-critical application because the delayed start of IEC tasks leads to a higher jitter. For further information, refer to the paragraph *Additional Information on the Option Configure synchronisation with IEC tasks...*, page 511.
	Compatibility Layout	Select this option to calculate the data output in the same way as in SoMachine / SoMachine Motion versions prior to V4.3. Do not use this layout together with exported `STRUCT`s that use the attributes *pack_mode* or *relative_offset*. The data layout created for the client is adapted as far as possible to the layout created by the compiler.

Element	Description	
	Optimized Layout	Select this option to calculate the data output in an optimized form, independent of the internal compiler layout. The optimization impacts only variables of a structured type and function blocks. No gaps with padding bytes are generated for members that are not published, for example, because they are deactivated in the **Symbol Configuration**. For internal members, for example, function blocks implementing interfaces, no gaps are created either. This option is by default selected for projects created with SoMachine / SoMachine Motion versions V4.3 or later, or with EcoStruxure Machine Expert. The setting is preserved after a **Project update**. This option requires SoMachine / SoMachine Motion versions V4.3 or later, or EcoStruxure Machine Expert. It is the default setting when creating a new **Symbol Configuration**.
	Use empty namespaces by default (V2 compatibility)	By default, this option is not selected, as EcoStruxure Machine Expert is delivered with an OPC server V3.5 or later. Select this option for the exchange of variables with an OPC server V2.0 to achieve the following behavior: • Program variables are exported without an application name (`Application.PLC_PRG.MyVar` –> `PLC_PRG.MyVar`). • Global variables are additionally exported without the GVL name (`Application.GVL.MyGlobVar` –> `.MyGlobVar`).
	Support Calls of Functions, FBs, Methods and Programs	This option is only available if the function is supported by the controller. Consult the *Programming Guide* specific to your controller for further information. If the option is selected, the action **Execute** can be set in the symbol table for symbols of POUs of type function, function block, method, or program. As a prerequisite, the **Support OPC UA features** option must be selected.
	Include call information in XML	This option is available if the option **Support Calls of Functions, FBs, Methods and Programs** is supported by the controller. Select this option to export information about called functions, function blocks, methods, or programs to the symbol file.
	Enable Symbol Sets	Select this option to display a toolbar above the **Symbol configuration** table. Also refer to the *Toolbar for Symbol Set Configuration* paragraph, page 509.
Tools button	**Save XML Scheme File...**	Opens the dialog box for saving a file in the file system. Allows you to create an XSD (XML Schema Definition) format of the symbol file, for use in external programs.

Description of the Table

Columns of the **Symbol configuration** table:

Column	Description
Symbols	The column shows a list of POUs. You can select variables to be exported. If you select variables of a structured data type, all members of the structure will be exported. You may also select only particular member variables in the dialog box **Symbol Configuration for Data Type**. Click the browse **...** button in the **Members** column to open this dialog box. For further information, refer to the description of the **Members** column.
Access Rights	To modify the access rights for a selected item, click the **Access Rights** column. Each mouse-click switches the symbol within the following definitions: • : read and write • : write-only • : read-only • none
Maximal	Shows the maximum access rights.
Attribute	An icon is displayed if the access right was assigned by attribute.
Type	Shows the data type of the variable. Variables of an alias data type are displayed as shown in the following example of a variable: `myVar : MY_INT;`, where `MY_INT` is an alias declared as: `TYPE MY_INT : INT; END_TYPE`. In this case, the **Type** column shows `MY_INT : INT`.
Members	Click the **...** button in the **Members** column to open the **Symbol Configuration for Data Type** dialog box. It allows you to select only particular member variables. In case of nested types, this dialog box again provides a button to open another **Symbol Configuration for Data Type** dialog box. This selection applies to all instances of this data type for which symbols are exported. If not all members of a structured type are exported, then an asterisk (*****) is displayed in the check boxes of the members to indicate that all exportable members of that type are exported.
Comment	Shows any comments which have been added in the declaration of the variable.

NOTE: With the POU property **Link Always**, an uncompiled POU can be forced to be downloaded to the controller. If this property is set in the **Build** tab of the **Properties** dialog box of the selected POU, then all variables declared in this POU are available, even if those variables are not referenced by other code. Alternatively, you can use the pragma {attribute linkalways}, page 614 to make not compiled variables available in the symbol configuration.

Variables which are configured to be exported but which are not valid in the application, for example because their declaration has been removed, are shown in red. This also applies to the concerned POU or library name.

By default, a symbol file is created with a code generation run. This file is transferred to the device with the next download. If you want to create the file without performing a download, use the command **Generate code**, by default available in the **Build** menu.

NOTE: Variables of a global variable list (GVL) will only be available in the symbol configuration if at least one of them is used in the programming code.

Toolbar for Symbol Set Configuration

If the option **Enable symbol sets** is selected in the settings of the symbol configuration, the toolbar for symbol set configuration is available with the following elements:

Element	Description
List	The list contains already defined symbol sets.
	Opens the **Add a New Symbol Set** dialog box for specifying a name for this set.
	Opens the **Add a Duplicate from Selected Symbol Set** dialog box. It allows you to create a copy of the set selected in the list. You can adapt the default name (*<group name>_duplicate*) to your individual requirements.
	Opens the **Rename the Selected New Symbol Set** dialog box for specifying another name for the set selected in the list.
	Opens a dialog box prompting whether the set selected in the list should be deleted.
Configure Symbol Rights	Opens the **Symbol Rights** view of the device editor, page 142.

Comments and Attributes Dialog Box

The **Comments and Attributes** dialog box opens upon clicking **Settings > Configure comments and attributes**. It contains the following elements:

Element	Description
Symbol table contents	
Enable extended OPC UA information	This function is not supported.
Include comments	
Include attributes	
Also include comments and attributes for type nodes	
XML symbol file contents	
Include namespace node flags	Namespace node flags provide additional information about the origin of a node in the namespace. Node flags are available in the symbol table when OPC UA is activated. Deselect this option to prevent namespace node flags from being inserted in the XML file if your parser cannot process them.
Include comments	Select this option to save comments in the XML file. In SoMachine / SoMachine Motion versions prior to V4.4, this includes the setting **Prefer docu comments**.
Include attributes	Select this option to save attributes in the XML file.
Also include comments and attributes for type nodes	This option is only available if the option **Include comments** or the option **Include attributes** is activated. If this option is selected, the information for type nodes is also included (user-defined types, such as STRUCT and ENUM elements). If this option is not selected, comments and attributes are only available for directly exported variables.
Select comments These parameters are only available if one of the **Include comments** options is activated.	
Include docu comments: **///They start with triple slash and are usually /// formatted in ReST (library documentation)**	Select the options to determine the kind of comments that are saved in the symbol configuration.
Include normal comments: **(*IEC/ Pascal style comments *) // C++-Style comments with double-slash**	
Always include both types of comments	
Prefer docu comments, fallback to normal ones	
Prefer normal comments, fallback to docu comments	
Filter Attributes (case insensitive) These parameters are only available if one of the **Include attributes** options is activated.	
Include all attributes (“foo”, “bar”, “foo.bar”)	Select the options to determine the attributes that are saved in the symbol configuration.
Match simple identifiers (“foo”, “bar”)	
Include attributes starting with:	
Filter Attributes with regular expression	

Additional Information on Configure synchronisation with IEC tasks...

To achieve synchronous, consistent access, the runtime system postpones the processing of read or write request of the symbolic client until no IEC task is executed. As soon as this gap is found, the restart of IEC tasks is postponed until the requested values have been copied to the variables list.

This option is useful for permanently running systems without production clocking, for example, if process values are to be written cyclically in fixed time intervals (such as 60 s).

NOTE: Do not activate the option **Configure synchronisation with IEC tasks...** for motion and real time-critical application because the delayed start of IEC tasks leads to a higher jitter.

If you intend to use the **Configure synchronisation with IEC tasks...** option, consider the following points when you define the variable lists that are read and written:

- Configure synchronous and consistent access only for those variables for which it is necessary.
- Create separate lists for consistent variables and for variables that may be inconsistent.
- Create several small lists containing consistent variables instead of one large list.
- Define the time intervals for cyclically reading values as large as possible.

The option **Configure synchronisation with IEC tasks...** is available at two different locations:

- In the **Symbol configuration** editor, as an option of the **Settings** button. (If a symbol configuration is available in the application.)
- In the **Options** tab of the **Properties** dialog box of the selected controller.

  NOTE: In order for the setting to take effect, perform a **Download** or **Online Change** of the applications on the controller and update the boot applications.

Symbol Configuration

Overview

The symbol configuration is used to create symbols, provided with specific access rights. They allow project variables to be accessed externally, for example by Vijeo-Designer, EcoStruxure Operator Terminal Expert, or OPC server.

Symbol Information

To save the symbols defined for an application as .xml file to a folder of your choice, run the
Project > Export Symbol Configuration... command (for further information, refer to the Menu Commands online help)
.

With EcoStruxure Machine Expert, the symbols defined for an application are exported to an XML file in the project directory (symbol file) when the application is downloaded to the controller. This file is named according to the following syntax:

<project name>.<device name>.<application name>.xml

Example: *proj_xy.PLC1.application.xml*

Symbols for input addresses and variables that are mapped to input channels allow read-only access. Write access is only allowed for testing purposes in simulation mode.

> **NOTE:** In case a download to the controller is not possible, you can create the symbol configuration file by executing the command **Generate code**.

Further on the symbol information is downloaded to the controller with the application. Depending on the device description, it will be included in the application or a separate child application will be generated. This will also be listed with the name *<application name>._symbols* in the Applications view of the device editor, page 122.

If the symbol configuration has been modified in online mode, you can reload it to the controller by clicking the button **Download** in the editor window, page 508.

For example, concerning the maximum number of applications on a controller, the symbol application has to be handled as a normal application.

If your controller has user management active, you can grant individual access rights to a symbol for different user groups (clients). To achieve this, place the same symbol in different symbol sets and allow or deny access to this symbol set for the individual user groups (clients).

Adding a Symbol Configuration

Prerequisites

Variables that will be exchanged between the controller and (multiple) HMI devices using the transparent Machine Expert protocol (see the *Machine Transparency* chapter in the *Introduction* User Guide) must be published in the controller using the **Symbol configuration**. They will then be available as variables in Vijeo-Designer, EcoStruxure Operator Terminal Expert, or OPC server.

⚠WARNING

UNINTENDED EQUIPMENT OPERATION

Use “naturally” aligned variables for exchange with the **Symbol configuration**.

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

For further information on “natural alignment”, refer to `Attribute pack_mode`, page 628.

Defining a Symbol Configuration

In order to get the symbol configuration functionality available, add the symbol configuration object to the application in the **Tools tree** as described in the *Opening the Symbol Configuration* paragraph. This IECVarAccess.library is automatically added to the **Library Manager**.

You can define the variables to be exported as symbols in the symbol configuration editor, page 505 or via pragmas (attribute symbol, page 631), which are to be added at the declaration of the variables.

NOTE: Variables of a global variable list (GVL) will only be available in the symbol configuration if at least one of them is used in the programming code.

NOTE: Constants are available in the **Symbol configuration** if the option **Replace constants** in the **Project Settings > Compile options** dialog box (see Menu Commands, Online Help) is deactivated. For particular constants, you can configure the option explicitly by using the attributes `{attribute 'const_replaced'}` and `{attribute 'const_non_replaced'}`.

However, in any case the following is true: Constants in a global variable list (GVL) or in another POU are only available in the **Symbol configuration**, if the GVL or POU is used in the program code, or if the compiler option **Link always** is activated in the **View > Properties > Build** dialog box (see Menu Commands, Online Help) of the GVL or POU.

Another possibility is provided by the SFC editor: You can define the implicitly created element flags in the element properties, page 335 for export to the symbol configuration.

The name of a symbol created by the symbol configuration is composed according to the following syntax:

<application name>.<POU name>.<variable name>

Examples:

MyApplication.PLC_PRG.a

MyApplication.GVL.a

For accessing the variable, define the symbol name completely.

Opening the Symbol Configuration

To open the **Symbol configuration**, proceed as follows:

Step	Action
1	Select the **Application** node in the **Tools tree**, click the green plus button, and select the command **Add other objects > Symbol configuration...**. **Result**: The **Add Symbol configuration** dialog box will be displayed.
2	In the **Add Symbol configuration** dialog box, optionally configure settings like whether to include comments or not. The settings can be modified later in the **Symbol configuration** editor, page 505.
3	Click the **Add** button. **Result**: A **Symbol configuration** node is created under the **Application** node in the **Tools tree**. The **Symbol configuration** is displayed on the right-hand side.

NOTE: Only one symbol configuration node can be created per device.

For details on the variables interchange between the controller and HMI part, refer to the chapter Data Exchange: Controller - HMIs, OPC DA and UA Server, page 517.

Creating Symbol Sets Allowing Different Access Rights for Different Control Clients

A symbol set is a defined set of symbols. If supported by the target device, you can combine different symbols of the application to form symbol sets in the symbol configuration editor. The symbol sets information transferred to the controller with the software download. You can grant access to each symbol set for the individual user groups. The access rights are assigned in the **Symbol Rights** view of the device editor, page 142.

Prerequisites for using symbol sets:

- A **Symbol Configuration** is available in the application.
- The **Enable symbol sets** option is enabled in the settings of the symbol configuration.
- The controller has a user management enabled.

  In the following example, a user group must be available that is granted the rights required for servicing the plant. The user group **Service** is by default available for this purpose.

To use symbol sets with OPC UA, a certificate is required. For further information on managing certificates for the OPC UA client, refer to the How To Manage Certificates with OPC UA Clients User Guide.

To create a symbol set and assign access rights, proceed as follows:

Step	Action	Comment
1	Open the **Communication Settings** tab of the device editor, page 87 to configure the connection to the controller.	–
2	Open the **Symbol Configuration** editor, and click the  button from the toolbar for symbol set configuration, page 509.	**Result**: The **Add a New Symbol Set** dialog box opens.
3	Enter a name for the set: `startup`.	–
4	In the toolbar of the **Add a New Symbol Set** dialog box, click the  button.	**Result**: The symbols available in the project are displayed.
5	Select the symbols you want to add to this set and save the project.	–
6	Click the **Configure Symbol Rights** button.	**Result**: The **Symbol Rights** view of the device editor, page 142 opens.
7	Click the **Synchronization**  button.	**Result**: The display of symbol sets is synchronized with the device. **NOTE:** If the user management is not yet enabled on the controller, a dialog box will be displayed, prompting you to do so.
8	Click **OK** to close the **Add a New Symbol Set** dialog box.	–
9	Select the **Users and Groups** view of the device editor, page 132, click the **Synchronization**  button, and click **Yes** to enable the user management.	**Result**: The **Device User Login** dialog box opens.
10	Log in. If you log in for the first time, execute the steps as described in the **Users and Groups** chapter, page 135.	**Result**: After successful login, the configuration of the device user management is displayed in the **Users and Groups** and **Access Rights** views, page 137.
11	Log in to the controller by executing the command **Online > Login** and click **Yes** to confirm that the application is downloaded to the controller.	–

Step	Action	Comment
12	After successful login, select the **Symbol Rights** view, and click the **Synchronization**  button.	**Results**: • The **Symbol sets** section displays the symbol sets that have been downloaded to the controller for the application (`startup` in this example). • The **Rights** table lists the user groups that are created in the user management of the controller (**Administrator** and **Service** in this example). • When a symbol set is selected on the left-hand side, the access rights of the individual user groups are displayed on the right-hand side: ◦ : access is granted ◦ : access is not granted The access allowed for each symbol is defined in the **Symbol Configuration** editor, page 505 (read, write, execute).
13	On the left-hand side, select the symbol set `startup`, and double-click the minus signs  for **Administrator** and **Service**.	**Results**: • The icons change into plus signs . • The **Administrator** and **Service** groups have access to the symbols in the `startup` symbol set.

You can download modifications performed on a symbol set definition to the controller by executing an online change, page 243. When the application is deleted on the controller, the symbol sets are also deleted. When building the application, you can create and save a symbol file in XML format for each symbol set.

Data Exchange: Controller - HMIs, OPC DA and UA Server

What's in This Chapter

Configuring Data Exchange of Variables 517
Publishing Variables in the Controller 521
Selecting Variables in the HMI 522
Publishing Variables in the HMI of Vijeo-Designer 523
Parameterization of the Physical Media 524
Data Exchange Performance Between Controller and HMI Configured with Vijeo-Designer 525
Indications for HMI Connections with Vijeo-Designer 527

Configuring Data Exchange of Variables

Overview

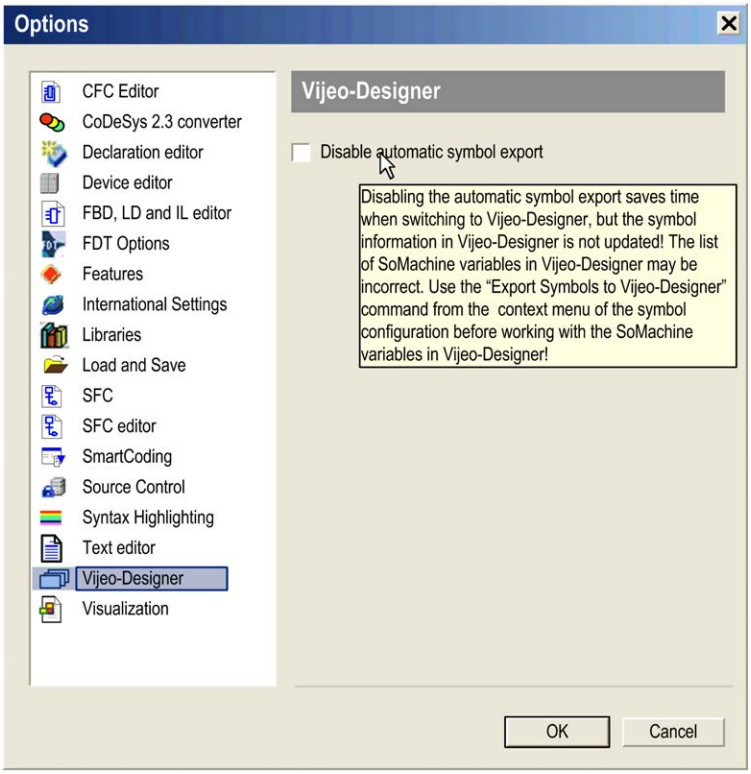
For configuring data exchange of variables from EcoStruxure Machine Expert to HMIs, perform the following steps:

Step	Action
1	Create variables in the controller part.
2	Publish the variables by defining them as **Symbols** in the controller part. **NOTE:** Variables that are published from EcoStruxure Machine Expert, are automatically available in Vijeo-Designer HMI applications (refer to the next section).
3	Configure the physical connection (automatically setup in EcoStruxure Machine Expert). **NOTE:** This step is not required in EcoStruxure Machine Expert for HMISCU controllers as they may communicate with their own control variables.

Disabling Automatic Symbol Export to Vijeo-Designer

By default, EcoStruxure Machine Expert automatically exports those variables defined as **Symbols** to the Vijeo-Designer HMI application.

Once symbols have been transferred to Vijeo-Designer, it is usually not necessary to make the transfer every time you call Vijeo-Designer. If you later add or modify symbols in your EcoStruxure Machine Expert application after having initially transferred the symbols, you can transfer symbols to Vijeo-Designer manually at will. To save time when you open Vijeo-Designer, you can disable the automatic transfer of symbols as follows:

Step	Action
1	Select the **Options...** command from the **Tools** menu. **Result**: The **Options** dialog box is displayed.
2	Select the entry **Vijeo-Designer** from the list on the left-hand side.
3	On the right-hand side, enable the check box **Disable automatic symbol export**. 
4	Click **OK** to close the dialog box.

NOTE: Activating the **Disable automatic symbol export** function inhibits the automatic export of EcoStruxure Machine Expert variables defined as **Symbols** to Vijeo-Designer. In order to perform this transfer manually, right-click the **Symbol configuration** node in the **Devices** window and execute the **Export Symbols to Vijeo-Designer** command. If you do not perform this manual transfer, Vijeo-Designer may not show the correct symbols which, in turn, may lead to errors being detected in the project.

⚠WARNING

UNINTENDED EQUIPMENT OPERATION

Execute the Export Symbols to Vijeo-Designer command if you have activated the Disable automatic symbol export before you start working in Vijeo-Designer.

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

Configuring Variable Data Exchange Manually

After you have published the variables in the **Symbol Configuration** editor, the variables must be exposed to the HMI tool (such as EcoStruxure Operator Terminal Expert or Vijeo Designer). Between EcoStruxure Machine Expert and Vijeo Designer, this procedure is by default performed automatically. To expose the variables to EcoStruxure Operator Terminal Expert or to Vijeo Designer if the option **Disable automatic symbol export** (see previous section) is activated, a manual import is required. Refer to the EcoStruxure Operator Terminal Expert or Vijeo Designer Online help for details on how to import variables.

NOTE: Before you start importing the variables manually, run the **Generate Code** command.

- The symbol file containing the symbols to be imported is placed in the same folder as the project with the name *<project>.<controller>.<Application>.xml*.

⚠WARNING

UNINTENDED EQUIPMENT OPERATION

Execute the Export Symbols to Vijeo-Designer command if you have activated the Disable automatic symbol export before you start working in Vijeo-Designer.

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

- To generate the symbol file and save it to a folder of your choice, run the command **Project > Export Symbol Configuration**.

  NOTE: Each modification to a variable in the **Symbol Configuration** requires a new import of the symbol file into the HMI tool.

  NOTE: For file-based storage *.fbs projects, the Vijeo Designer content is stored as a binary file that is not editable.

Variable Types for EcoStruxure Machine Expert - HMI Data Exchange

The following table lists the variable types for EcoStruxure Machine Expert - HMI data exchange:

Variable type EcoStruxure Machine Expert	Variable type Vijeo-Designer	Variable type EcoStruxure Operator Terminal Expert	Comment
BOOL	BOOL	BOOL	--
BYTE	Integer	BYTE	--
WORD	UINT	WORD	--
DWORD	UDINT	DWORD	--
SINT	Integer	SINT	--
INT	INT	INT	--
DINT	DINT	DINT	--
USINT	Integer	USINT	--
UINT	UINT	UINT	--
UDINT	UDINT	UDINT	--
LINT	–	LINT	--
ULINT	–	ULINT	--
REAL	REAL	REAL	--
LREAL	–	LREAL	--
STRING	STRING	STRING	--
WSTRING	STRING	WSTRING	Vijeo-Designer supports WSTRING as a general STRING type. Therefore, you can either exchange only STRINGS or only WSTRINGS with the HMI. A mixture of these two variable types is not allowed. If you use WSTRINGs, all your strings must be WSTRINGs. Indicate to the Vijeo-Designer driver that all strings should be managed as UNICODE WSTRINGs as follows: Select the node **SoMachineNetwork** or **SoMachineCombo** in the **Navigator** tree of Vijeo-Designer and set the parameter **String Encoding** to the value **Unicode**. EcoStruxure Operator Terminal Expert can use STRING and WSTRING simultaneously.
ARRAY	–	–	In Vijeo-Designer and EcoStruxure Operator Terminal Expert, you can only reference the elements of an array, not the whole array. **Example**: Your array consists of SINTs called `myValues`. You can reference `myValues[0]` or `myValues[5]` and put this into a control on the HMI screen. Arrays must not contain more than 2,048 elements. If you try to use arrays with more than 2,048 elements in Vijeo-Designer, a message is issued.
Non-zero based array	–	Non-zero based array	In Vijeo-Designer, you cannot import an array that is defined, example: `myArray[1..100]`.
ARRAY OF ARRAY	–	–	You cannot export to Vijeo-Designer an array that has an array as its element type, such as `ARRAY [0..9] OF ARRAY [0..9] OF INT`. Nevertheless,

Variable type EcoStruxure Machine Expert	Variable type Vijeo-Designer	Variable type EcoStruxure Operator Terminal Expert	Comment
			you can use multi-dimensional arrays, such as `ARRAY [0..9, 0..9] OF INT`.
DUT	–	–	In Vijeo-Designer and EcoStruxure Operator Terminal Expert, you can only reference the elements of a DUT, not the whole DUT. This behavior is similar to the behavior of arrays.
TIME	–	TIME	–
DATE	–	DATE	–
TOD	–	TOD	–
DT	–	DT	–
REFERENCE TO <Type>	<Type> (<Type> being one of the supported types in this list)]	<Type> (<Type> being one of the supported types in this list)]	Do not share references to ARRAYs or structured variables containing an ARRAY in the symbol configuration editor, as their values will not be displayed correctly on the HMI.

NOTE: The variables from the *PLC_R* structures of the PLCSystem library cannot be shared via the **Symbol Configuration** with the Vijeo-Designer and EcoStruxure Operator Terminal Expert application of HMI targets (including HMI controllers).

For further information on variable types for EcoStruxure Machine Expert - HMI data exchange, see the Vijeo-Designer and EcoStruxure Operator Terminal Expert online help.

Identifier Length

In Vijeo-Designer, the maximum length of the Symbol name is limited to 32 characters.

Variable names longer than 32 characters produce an error message and are not imported.

Publishing Variables in the Controller

Overview

Publish variables in the controller part of the EcoStruxure Machine Expert application within the **Symbol configuration** editor.

⚠WARNING

UNINTENDED EQUIPMENT OPERATION

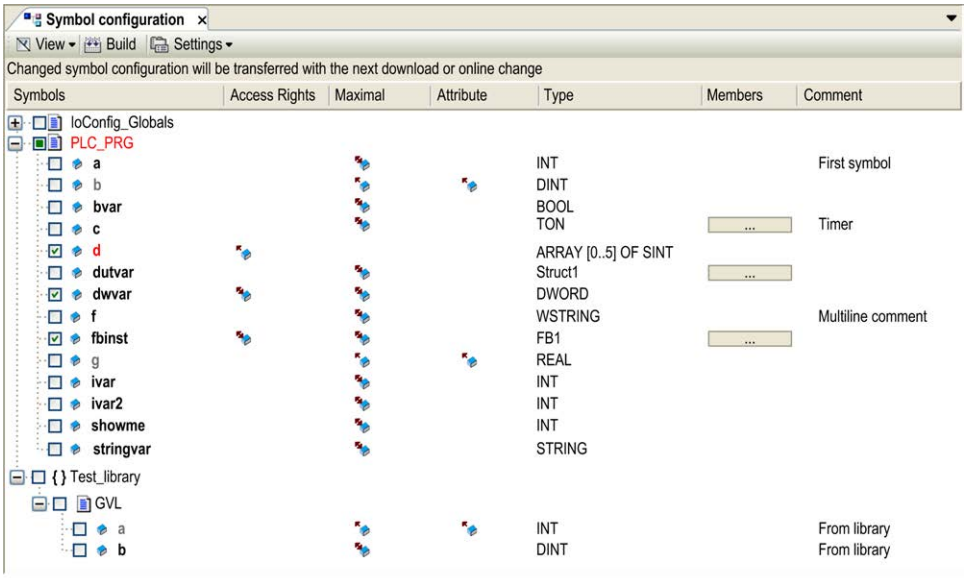
Use “naturally” aligned variables for exchange with the **Symbol configuration**.

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

For further information on “natural alignment”, refer to `Attribute pack_mode`, page 628.

Publishing Variables in the Symbol Configuration Editor

To publish variables within the **Symbol configuration** editor, proceed as follows:

Step	Action
1	Create a **Symbol Configuration** node under the **Application** node in the **Tools tree** as described in the *Adding a Symbol Configuration* chapter, page 512.
2	Double-click the **Symbol Configuration** node to open the **Symbol configuration** editor.
3	In the **Symbol configuration** editor, select those elementary variables that you wish to publish for communication with 1 or several HMI terminals by selecting or deselecting the check box in the **Symbols** column:  You can also assign read/write access rights to each variable individually in the **Access Rights** column. For further information, refer to the description of the Symbol configuration editor, page 505. For using symbol sets, refer to Creating Symbol Sets Allowing Different Access Rights for Different Control Clients, page 514. **Note**: Only variables on elementary data types, page 635 are available for interchange with HMI terminals.
4	For your settings to become valid, click the **Download** link in the **Symbol configuration** editor.

NOTE: The publishing mechanism consumes an overhead of about 50 Kbyte in the controller. Each published variable consumes 11 bytes within the controller application.

Selecting Variables in the HMI

Selecting Variables

With EcoStruxure Machine Expert, the variables that have been published in the controller are directly available in Vijeo-Designer.

EcoStruxure Operator Terminal Expert: For information, refer to the EcoStruxure Operator Terminal Expert online help.

In the **Expression Editor Pad** of Vijeo-Designer, select the corresponding tab to have direct access to the variables published in EcoStruxure Machine Expert.

For further information, refer to the Vijeo-Designer online help.

Publishing Variables in the HMI of Vijeo-Designer

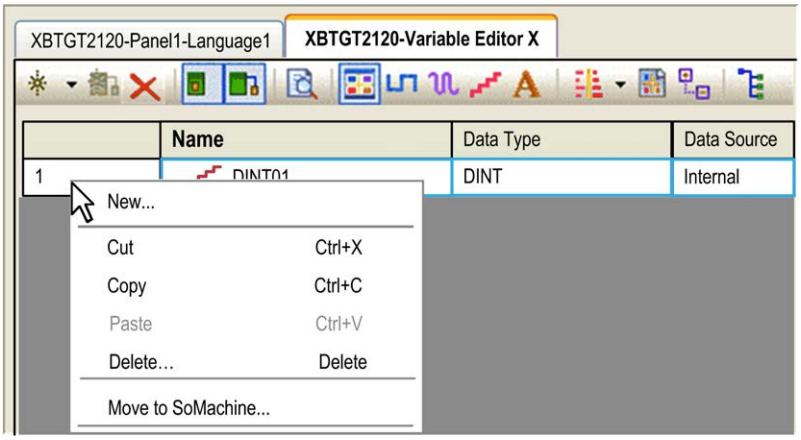
Supported Variable Types

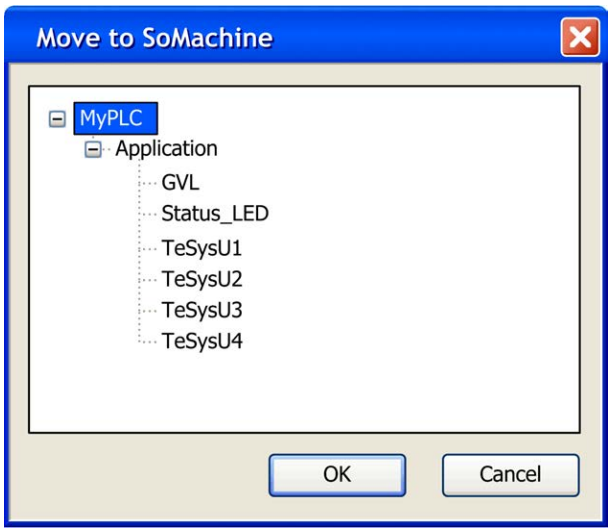
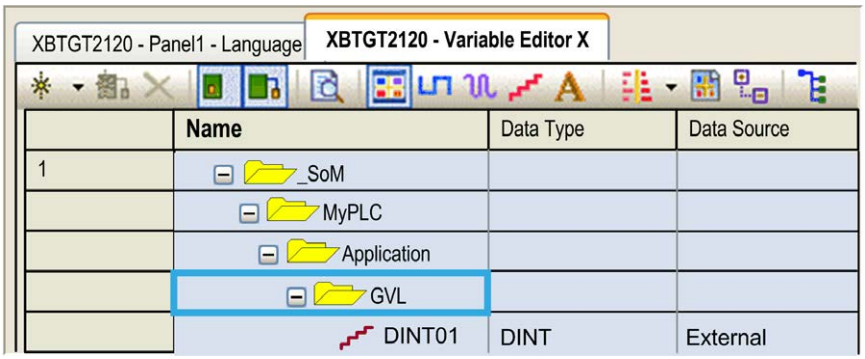
The following variable types can be published in Vijeo-Designer to make them available for the entire EcoStruxure Machine Expert project:

- BOOL
- DINT
- INT
- UINT
- UDINT
- Integer
- REAL
- STRING

Procedure

To publish the above mentioned variable types, proceed as follows:

Step	Action
1	In the Vijeo-Designer **Variable Editor**, select those variables you want to publish.
2	Right-click the selected variable(s) and execute the command **Move to SoMachine** from the contextual menu.  **Result**: The **Move to SoMachine** dialog box will be displayed.

Step	Action
3	In the **Move to SoMachine** dialog box open the subfolders of the devices defined in EcoStruxure Machine Expert to see the levels where variables are defined (POU or GVL).
4	Select the POU or GVL to which you want to add the selected Vijeo-Designer variable(s) and click **OK**.  **Result**: The selected variable(s) has / have been moved to the selected EcoStruxure Machine Expert POU or GVL and is / are available throughout the EcoStruxure Machine Expert project. 

Parameterization of the Physical Media

Overview

The runtime data exchange between the controller and the HMI is executed on different media, depending on the selected hardware. For further information, refer to the Online Help of Vijeo-Designer or EcoStruxure Operator Terminal Expert. This section provides an example using Vijeo-Designer.

Configuration Example for Vijeo-Designer

The default settings below are valid for communications between M241 and an HMI panel via serial line RS-485 using an XBTZ9008 cable (serial line SubD-RJ45).

Configuration of M241 with HMI panel:

M241 controller serial line configuration

Parameter	Value
Physical Medium	RS485
Baud rate	115200
Parity	none
Data bits	8
Stop bits	1

HMI panel IO-Manager configuration using a driver: EcoStruxure Machine Expert - network with at least one Scan-Group (for further information, refer to the Vijeo-Designer online help).

Parameter	Value
Physical Medium	RS485
Baud rate	115200
Parity	none
Data bits	8
Stop bits	1
Equipment Name	Controller device name (available in the **Communication Settings** tab, page 87).

Configuration of XBTGC:

Device	Configuration
XBTGC controller	No configuration required.
HMI panel	IO-Manager
Driver	EcoStruxure Machine Expert - Combo with at least one Scan-Group

Data Exchange Performance Between Controller and HMI Configured with Vijeo-Designer

Overview

The communication speed between controller and HMI depends largely on the number of variables that are exchanged. Therefore, the time that is required to display the values on the HMI panels when a controller-HMI-connection is established, as well as to the refresh time of the variables, are affected accordingly.

This chapter provides reference values that have been achieved under optimum conditions. Actual values depend on the total performance of your controller application (for example, the communication task responsible for data exchange is executed with a low priority).

For data exchange using the Machine Expert protocol via Ethernet, this chapter indicates the number of variables allowed to achieve a good data transmission performance. If you are using serial line, consider to change to Ethernet for increasing the performance.

General Measures for Improving Communication Performance

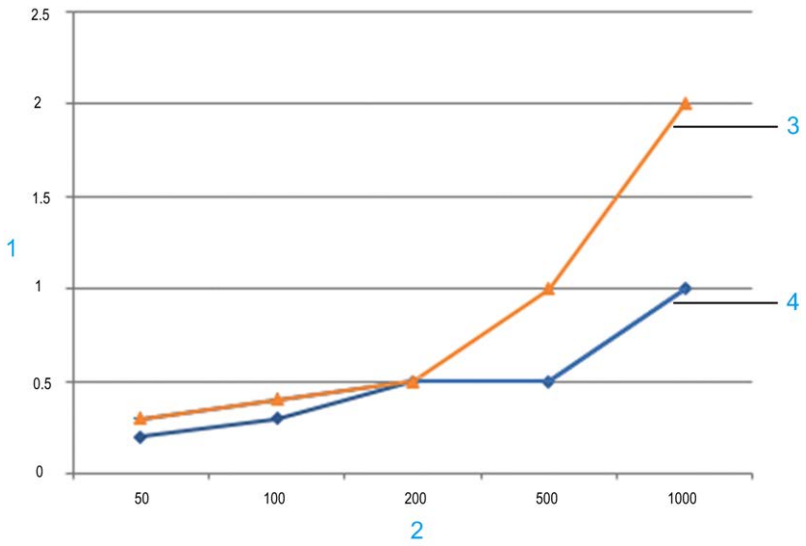
To improve the communication performance, you can take the following measures:

- In the equipment or scan group properties of your HMI, set the Vijeo-Designer parameter **ScanRate** to **Fast**.
- Reduce the number of variables per HMI panel because only the variables on the active panel are refreshed. It is a good practice to create several HMI panels with reduced number of variables in Vijeo-Designer instead of creating one HMI panel that shows many variables.
- Add only those variables to the **Symbol configuration** that are used in the HMI.

Variable-to-Time Ratio for Refreshing Variables on the HMI Panel

The graph indicates reference values that have been measured for the time that is required to refresh variables over the Machine Expert protocol via Ethernet between XBTGT HMI and M••• controllers with different numbers of variables under non-industrial conditions.

Typical delay to refresh variables on the HMI:



1 Time in seconds

2 Number of variables

3 XBTGT2330 + M••• controller

4 XBTGT4330 + M••• controller

Vijeo-Designer Suggestions on Variables

Vijeo-Designer provides the following suggested guidance for using variables in the Vijeo-Designer online help:

Chapter *Creating Variables → About Variables and Device Addresses → Source: Internal Versus External*:

- One target can have a maximum of 8000 or 12000 variables depending on the target type. Array and structure holders (the group node) also count as variables. A block variable counts as one variable.
- You can use a maximum of 800 variables on a single panel.

Chapter *Appendix → Run-Time Specifications*:

Number of variables per panel (limit):

Controller	Maximum number of variables per panel
iPC series	2500
Other target types, except iPC	800

Number of variables per target (limit):

Controller	Maximum number of variables
• iPC* • XBTGTW series	12000
• XBTGC • XBTGT • XBTGH • HMIGTO • HMISTO • HMISTU • HMISCU series	8000
XBTGK series	8000

*	For iPC: If persistent variables, such as alarm variables and data logging variables, are used, a maximum of 8000 variables can be supported for each iPC target.

Chapter *Errors → Message List → Editor Error Messages→ 1300 - 1999→ Error 1301*:

Error 1301: [Target] [target name] too many variables. Variable limit is [8000 or 12000].

NOTE: The Vijeo-Designer online help indicates that the total number of elements in an array must not exceed 2048 (refer to the chapter *Creating Variables → Array Variables*). This limits the size of (single or multidimensional) array variables that are shared via the EcoStruxure Machine Expert **Symbol Configuration**. To overcome this limit, consider sharing an array of DUT (for example, ARRAY[0..99] OF DUT_30, where DUT_30 is a user-defined type containing 30 distinct INT variables, resulting in 3000 variables). In any case, the Error 1301 will be issued if the maximum number of variables per target (8000 or 12000) is exceeded.

Indications for HMI Connections with Vijeo-Designer

Indications Regarding Communication Settings View

When you connect controllers to HMIs, select the option **Nodename** from the **Connection Mode** list in the **Communication Settings** view, page 100 to specify the address. Otherwise, a corresponding message will be generated as soon as you intend to download a project to the HMI.

Indications for Windows Embedded HMIs

For Windows embedded HMIs, verify the following:

- Make sure that the Windows service CODESYS Control is running on the Windows Embedded HMI.
- Start the Vijeo-Designer Runtime to be able to download the application (for further information, refer to the Vijeo-Designer online help, chapter *25.7 Run the User Application*).

Cam Motion Editor

What's in This Chapter

Cam Motion Editor - General Information 529
Generating IEC Program Code from Cam Data 530
Online View and Functions of the Cam Motion Editor 532
Discontinuous Courses of the Positions 533
Dialog Boxes 534

Cam Motion Editor - General Information

General Information

Overview

The cam motion editor is used to create and parameterize cam diagrams and assign them to *FB_MultiCam* / *MC_Cam_ID* function blocks. In turn, these diagrams can then be used by the Motion Sizer, for example, for drive sizing.

The cam diagrams are based on the motion laws defined in VDI 2143. They allow you to design the motion of your axis in relation to the position of a master axis φ. Cam diagrams are used to model the movement of a servo axis, which follows the motion of a virtual line shaft. The angle of this line shaft is described by φ.

You can define segments for each cam diagram that correspond to the different sequences of motion that your axis performs.

When you modify the configuration in the **Configuration** Tab, page 545, test your application before you enable the online mode (also refer to Online View and Functions of the Cam Motion Editor, page 532).

⚠WARNING

UNINTENDED EQUIPMENT OPERATION

Thoroughly test your application code for proper operation before placing your system into service.

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

Adding a Cam Object

Overview

To add a cam diagram, proceed as follows:

Step	Action
1	In EcoStruxure Machine Expert, right-click the **Application** node in the **Tools tree** and select **Add Object > CamDiagram (classic)...**
2	In the **Add CamDiagram (classic)** dialog box, enter a **Name** for the new cam diagram.
3	Click **Add** to add the cam object. As a result, a **Camdiagram** node with the defined name is created below the **Application** node. By default, it contains a **Segment** subnode that is a child object of the diagram object.

Opening the Motion Editor of a Cam Object

Procedure

To open the motion editor of a cam diagram or a cam segment, double-click the node in the **Tools tree**:

If...	Then ...
You double-click the **CamDiagram** node	The **Motion Editor** tab opens in the multi-tabbed editor view.
You double-click the **Segment** node	The **Motion Editor** tab opens in the multi-tabbed editor view and highlights the selected segment with a green background in the diagram.

Displaying More than one Motion Editor

To compare interdependent motions with each other, you can display several motion editor views simultaneously. To achieve this, execute the command **Window > New Horizontal Tab Group** or **Window > New Vertical Tab Group**.

Generating IEC Program Code from Cam Data

Using Cam Data for Function Blocks

Overview

Cam diagrams are used as a basis to generate cam data structures with a corresponding number of cam points.

These data structures can be used to parameterize the *FB_MultiCam / MC_Cam_ID* function blocks or a POU with equivalent functionality.

Cam data is used to initialize function blocks/modules that are processing cam data structures in the application program.

Specifying the Target Function Block

The **Configuration** tab of the cam motion editor provides an input field that allows you to specify an existing cam data structure as target. If a target is defined, cam data will automatically be written to this target structure when the program is initialized. If the expression is indeterminable, a diagnostic message will be generated.

From the target defined, EcoStruxure Machine Expert detects whether the data structures are CommonMotionType or PacDriveLib structures and parameterization is performed accordingly.

Data Structure	Library
ST_Multicam	PacDriveLib
FB_MultiCam / MC_Cam_ID	CommonMotionType

Generating Data for Cam Data Structures

When the project is built (using the **Generate Code** command from the **Build** menu or shortcut **F11**), IEC source code is generated and written into the specified cam data structure.

The parameterization statements for the cam data structure are generated in the following format:

```
MyMultiCam.MultiCamData.diNumberOfCamPoints := 3;
MyMultiCam.MultiCamData.lrYPeriod := 0;
MyMultiCam.MultiCamData.astCamPoint[0].lrX := 0;
MyMultiCam.MultiCamData.astCamPoint[0].lrY := 100;
MyMultiCam.MultiCamData.astCamPoint[0].lrM := 0;
MyMultiCam.MultiCamData.astCamPoint[0].lrK := 0;
MyMultiCam.MultiCamData.astCamPoint[0].lrLambda := 0.5;
MyMultiCam.MultiCamData.astCamPoint[0].lrC := 1;
MyMultiCam.MultiCamData.astCamPoint[0].etCamType := ET_
CAMTYPE.Poly5Com;
MyMultiCam.MultiCamData.astCamPoint[1].lrX := 100;
MyMultiCam.MultiCamData.astCamPoint[1].lrY := 0;
MyMultiCam.MultiCamData.astCamPoint[1].lrM := 0;
MyMultiCam.MultiCamData.astCamPoint[1].lrK := 0;
MyMultiCam.MultiCamData.astCamPoint[1].lrLambda := 0.5;
MyMultiCam.MultiCamData.astCamPoint[1].lrC := 1;
MyMultiCam.MultiCamData.astCamPoint[1].etCamType := ET_
CAMTYPE.SimplSin;
```

In this example, `MyMultiCam.MultiCamData` is configured as target data structure.

The generated data is based on cam diagrams that were edited in offline mode. They are not updated in online mode.

Maximum Number of CamPoints

Each cam segment is defined by a CamPoint that specifies the data located on the left margin of the segment. The right CamPoint of a cam segment corresponds to the left CamPoint of the next cam segment. To identify the end point of the cam diagram, the last cam segment is defined by an own right CamPoint.

There is a maximum of 32 CamPoints available for a cam data structure. When a target structure has been selected with a cam diagram having more than 31 segments, a diagnostic message will be generated. In order to design the motion of an axis with more than 31 segments, you must divide it into several cam diagrams.

> **NOTE:** If several cam diagrams are incorporated for drive sizing with Motion Sizer, Motion Sizer ensure that the total cam curve including all sub cam diagrams is used.

Initializing the Function Block

In order to provide the configured cam data to the target structure, an initialization code is executed once during start-up. This initialization task is executed by a program POU that is called up as initialization code after the user initialization processes have been completed.

Copying the Source Code of the Cam Diagram

Overview

The **Generated IEC source code** tab of the cam motion editor provides the source code that has been generated for the cam diagram in a read-only format. Whenever the cam diagram is modified, the code is regenerated.

You can copy the IEC code for further usage in your program.

Online View and Functions of the Cam Motion Editor

Online View and Functions of the Cam Motion Editor

Cam Motion Editor in Online Mode

If you enable the online mode while the cam motion editor is active/visible, then the cam motion editor is also switched to online mode if a cam data structure is configured as target, page 545. If no target structure is configured, the online mode is not available and the originally configured diagram is displayed.

As long as the online mode is enabled and a cam data structure is defined as target, the values and graphs displayed in the **Motion Editor** tab represent the online values of the selected target structure.

You can select a segment, but you cannot modify anything:

- You cannot modify the segment data.
- You cannot specify a variable as target.
- You cannot modify the axis-diagram allocation.

  NOTE: Some, especially user-defined, profiles are not supported by the editor. This is indicated by segments being grayed and position, velocity, acceleration, jerk lines not being displayed.

Online Change

The cam motion editor supports the **Online change** function, page 243.

After a cam data structure has been downloaded to the controller, the individual segments will be editable as soon as you have switched to offline mode.

When switching back to online mode, the modifications are detected and you will be prompted to decide whether you want to log in and perform the online change, page 241. If you confirm this message with **OK**, the cam data structure will be updated.

For the modifications to become effective, the `FB_MultiCam` / `MC_Cam_ID` function blocks must be switched to the adequate mode to take into account the modified cam data structure.

You can achieve this by setting the input parameter (for example, `iq_xNewCam`...) via the **Debug > Force values** command. For further information, refer to the description of *FB_MultiCam* in the PD_PacDriveLib or *FB_MultiCam / MC_Cam_ID* in the CommonMotionType library guide.

Discontinuous Courses of the Positions

Discontinuous Courses of the Positions

Overview

The cam data structure is used to parameterize the *FB_MultiCam* / *MC_Cam_ID* function blocks. The data structure cannot map any discontinuous courses of the positions. Therefore, the *FB_MultiCam* / *MC_Cam_ID* function blocks do not move along inconsistently parameterized cam courses.

In order to help avoid invalid parameterization, the positions are verified for inconsistencies.

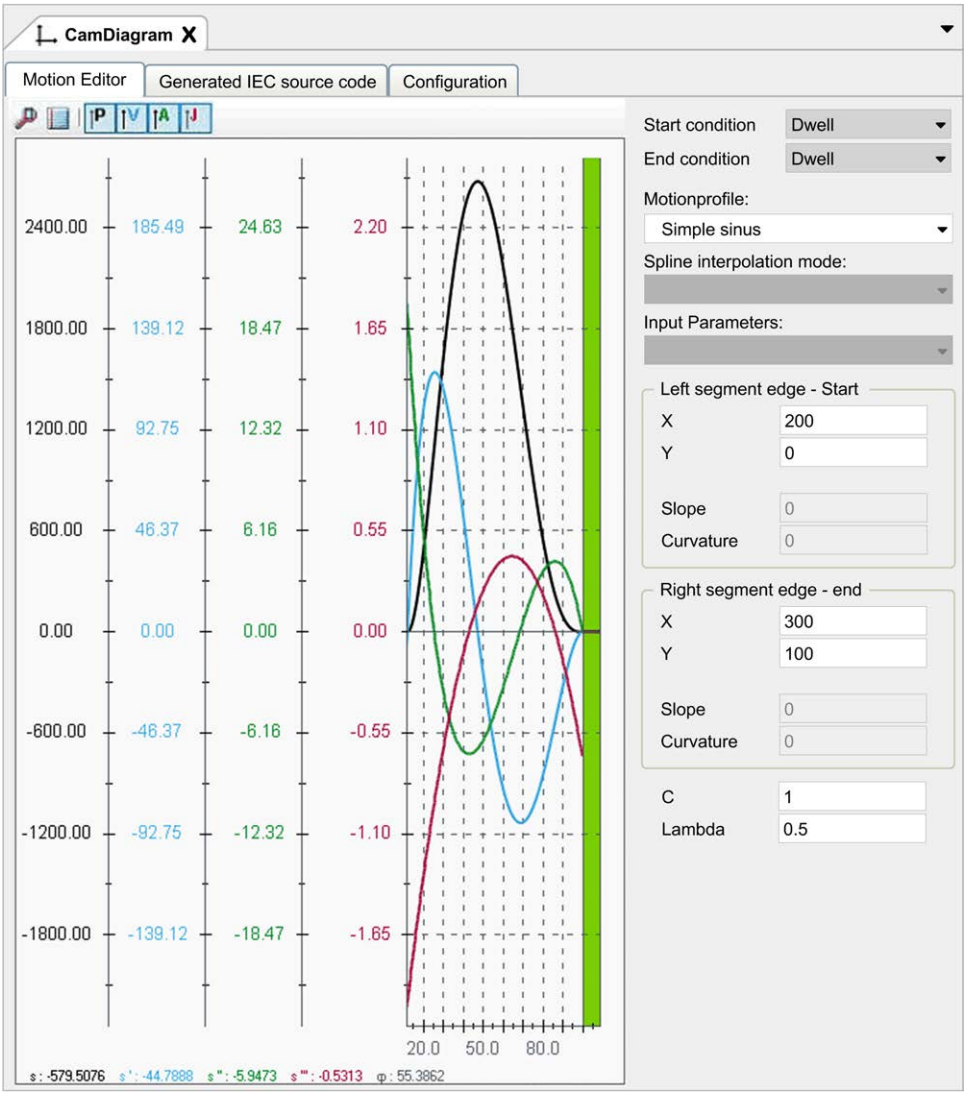
For this purpose, the start point of a cam segment is compared to the end point of the preceding segment. For each invalid parameterization, a compile error is generated.

Dialog Boxes

Motion Editor

Motion Editor Tab

The graphic displays the **Motion Editor** tab.



Element	Description	Value range
Cam diagram	The view on the left-hand side of this tab displays the individual segments of the cam diagram that have been double-clicked in the **Tools tree**. The selected cam segment is highlighted green. To select a cam segment, double-click the segment in the **Tools tree** view or click the segment in the **Motion Editor** tab.	–
Start condition	The condition that applies at the start point of the selected cam segment.	• **Dwell** • **Velocity** • **Return** • **Motion**
End condition	The condition that applies at the end point of the selected cam segment.	• **Dwell** • **Velocity** • **Return** • **Motion**
Motionprofile	The motion law describes the profile of the segment. The available motion profile depends on the **Start condition** and **End condition** selected. Refer to the table *Motion Profiles Available for Combinations of Start and End Conditions*, page 538.	• **Straight line** • **Quadratic parabola** • **Polynomial of 5th degree** • **Simple sinus** • **Modified sinus** • **Modified acceleration trapezoid** • **General polynomial of the 5th degree** • **Sinus-straight line combination** • **Sloped sinus** • **Modified sinus combination** • **Modified acceleration trapezoid combination** • **Harmonic combination** • User-defined motion profiles
Spline interpolation mode	This parameter is only available for user-defined motion profiles. It defines how the edge condition at the start point and end point of the profile is resolved.	• **Natural spline** • **Spline on base of the slope of the borders** • **Periodic spline**
Input parameters	Select from the list the parameters which you will configure. **Result**: The corresponding text fields will become editable. The remaining parameter will be calculated by EcoStruxure Machine Expert.	The values listed in the table depend on the **Motionprofile** you selected. Refer to the following tables that list the parameters that are available per motion law.
Left segment edge - Start section		
X	The x coordinate of the start point of the selected cam segment. Represents the angle of the virtual line shaft where the segment starts.	Decimal number Unit: φ
Y	The y coordinate of the start point of the selected cam segment. Represents the position of the axis at the segment start. This is the value of s at the position x.	Decimal number Unit: Position
Slope	The slope of the position curve at the start point of the selected cam segment. This is the value of s'.	Decimal number Unit: Position / φ
Curvature	The curvature of the position curve at the start point of the selected cam segment. This is the value of s''.	Decimal number Unit: Position / φ^2
Right segment edge - end section		

Element	Description	Value range
X	The x coordinate of the end point of the selected cam segment. Represents the angle of the virtual line shaft where the segment ends.	Decimal number Unit: φ
Y	The y coordinate of the end point of the selected cam segment. Represents the position of the axis at the segment end. This is the value of s at the position x.	Decimal number Unit: Position
Slope	The slope of the position curve at the end point of the selected cam segment. This is the value of s'.	Decimal number Unit: Position / φ
Curvature	The curvature of the position curve at the end point of the selected cam segment. This is the value of s''.	Decimal number Unit: Position / $φ^2$
C	Defines the fraction of the selected cam segment that is spent for changing the velocity. The value 1 indicates that there is no constant velocity phase within the segment. Acceleration and deceleration procedures are performed throughout the segment. The value 0.5, for example, indicates that acceleration and deceleration procedures are performed during the half of the segment time. The other half of the segment, the inflexion point between acceleration and deceleration, is reserved for constant velocity.	0...1
Lambda	Defines the position of the inflexion point (when **C**=1) or of the phase within constant velocity (when **C**<1) on the x axis (φ) within the selected cam segment. **Lambda** describes the proportion between acceleration and deceleration on the virtual line shaft angle within the segment. In a segment that only consists of an acceleration phase (**Start condition** = **Dwell**, **End condition** = **Velocity**, the **Lambda** value is fixed to 1. Therefore, 100% of the time of changing the velocity is used for acceleration, 0% is used for deceleration. In a segment that only consists of a deceleration phase (**Start condition** = **Velocity**, **End condition** = **Dwell**, the **Lambda** value is fixed to 0. Therefore, 0% of the time of changing the velocity is used for acceleration, 100% is used for deceleration. In a segment that only consists of acceleration and deceleration phases (**Start condition** = **Dwell**, **End condition** = **Dwell**, the **Lambda** value is greater than 0 and less than 1. A value of 0.5 means that acceleration and deceleration phases have the same length on the x axis (φ). 50% of the total length is spent with changing the velocity. A value of 0.1 means that 10% of the acceleration and deceleration length on the x axis (φ) are used for deceleration. 90% are used for acceleration.	0...1

Additional information for certain motion laws:

Designation	Description
C	Cam percentage of a motion curve. Valid interval: The value of the input field must be greater than 0 and less than or equal to 1. Examples: **C** = 1: The course is bent or a cam. **C** = 0.001: The course almost matches a straight line completely. **C** = 0.4: The course has a straight line percentage of 60% and a cam percentage of 40%.
Lambda	Position of the inflexion point. For dwell/dwell laws the inflexion point is within the valid interval: The value of the input field must be greater than or equal to 0 and less than or equal to 1. For **Lambda** = 0.5, the inflexion point is precisely in the center of the cam segment. For **Lambda** = 0.00001, the inflexion point is almost at the left edge. For the most dwell/velocity or velocity/dwell motion laws, **Lambda** is exactly 1 or 0 and is precisely at the left or right edge.

Motion Profiles Available for Combinations of Start and End Conditions

Depending on the combination of the boundary conditions (start condition, end condition) you can select the following **Motionprofiles**.

<table>
<tr><th>From/to</th><th>Dwell
(v=0, a=0)</th><th>Velocity
(v<>0, a=0)</th><th>Return
(v=0, a<>0)</th><th>Motion
(v<>0, a<>0)</th></tr>
<tr><td>Dwell
(v=0, a=0)</td><td>Straight line
Quadratic parabola
Polynomial of 5th degree
Simple sinus
Modified sinus
Sloped sinus
Modified acceleration trapezoid
General polynomial of the 5th degree</td><td>Quadratic parabola
Polynomial of 5th degree
Simple sinus
Modified sinus
Modified sinus combination
Modified acceleration trapezoid
General polynomial of the 5th degree
(Lambda = 1)</td><td>General polynomial of the 5th degree
Modified acceleration trapezoid combination
Harmonic combination</td><td>General polynomial of the 5th degree</td></tr>
<tr><td>Velocity
(v<>0, a=0)</td><td>Quadratic parabola
Polynomial of 5th degree
Simple sinus
Modified sinus
Modified sinus combination
Modified acceleration trapezoid
General polynomial of the 5th degree
(Lambda = 0)</td><td>Straight line
General polynomial of the 5th degree
Modified sinus combination</td><td>Harmonic combination
General polynomial of the 5th degree</td><td>General polynomial of the 5th degree</td></tr>
<tr><td rowspan="2">Return
(v=0, a<>0)</td><td>Modified acceleration trapezoid combination
Harmonic combination</td><td>Harmonic combination</td><td rowspan="2">Sinus straight combination
General polynomial of the 5th degree</td><td rowspan="2">General polynomial of the 5th degree</td></tr>
<tr><td colspan="2">General polynomial of the 5th degree</td></tr>
<tr><td>Motion
(v<>0, a<>0)</td><td colspan="4">General polynomial of the 5th degree</td></tr>
</table>

Parameters for Motion Laws Quadratic parabola, Polynomial of 5th degree, Simple sinus, Modified sinus, Modified acceleration trapezoid

The table lists the parameters that are available for the motion laws **Quadratic parabola**, **Polynomial of 5th degree**, **Simple sinus**, **Modified sinus**, **Modified acceleration trapezoid**, and the values they can have in accordance with the selected **Start condition** and **End condition**:

Parameter	**Start condition - End condition**		
	Dwell - Dwell	**Dwell - Velocity**	**Velocity - Dwell**
X start	User input	User input	User input
Y start	User input	User input* or calculated by EcoStruxure Machine Expert	User input* or calculated by EcoStruxure Machine Expert
Slope start	Fixed to 0	Fixed to 0	User input* or calculated by EcoStruxure Machine Expert
Curvature start	Fixed to 0	Fixed to 0	Fixed to 0
X end	User input	User input	User input
Y end	User input	User input* or calculated by EcoStruxure Machine Expert	User input* or calculated by EcoStruxure Machine Expert.
Slope end	Fixed to 0	User input* or calculated by EcoStruxure Machine Expert.	Fixed to 0
Curvature end	Fixed to 0	Fixed to 0	Fixed to 0
C	User input(0...1)	User input* (0...1) or calculated by EcoStruxure Machine Expert.	User input* (0...1) or calculated by EcoStruxure Machine Expert.
Lambda	User input(0...1)	Fixed to 1	Fixed to 0
* By default, one out of four parameters is calculated by EcoStruxure Machine Expert, whereas the other three parameters are editable. You can change this by clicking the option next to the parameter.			

Parameters for Motion Law Sloped sinus

The table lists the parameters that are available for the motion law **Sloped sinus** and the values they can have in accordance with the selected **Start condition** and **End condition**:

Parameter	**Start condition - End condition**
	Dwell - Dwell
X start	User input
Y start	User input
Slope start	Fixed to 0
Curvature start	Fixed to 0
X end	User input
Y end	User input
Slope end	Fixed to 0
Curvature end	Fixed to 0
C	User input(0...1)
Lambda	User input(0...1)

Parameters for Motion Law Modified sinus combination

The table lists the parameters that are available for the motion law **Modified sinus combination** and the values they can have in accordance with the selected **Start condition** and **End condition**:

Parameter	**Start condition - End condition**		
	Velocity - Velocity	**Dwell - Velocity**	**Velocity - Dwell**
X start	User input	User input	User input
Y start	User input	User input	User input
Slope start	User input	Fixed to 0	User input
Curvature start	Fixed to 0	Fixed to 0	Fixed to 0
X end	User input	User input	User input
Y end	User input	User input	User input
Slope end	User input	User input	Fixed to 0
Curvature end	Fixed to 0	Fixed to 0	Fixed to 0
Lambda	User input(0...1)	User input(0...1)	User input(0...1)

Parameters for Motion Law Straight line

The table lists the parameters that are available for the motion law **Straight line** and the values they can have in accordance with the selected **Start condition** and **End condition**:

Parameter	**Start condition - End condition**	
	Dwell - Dwell	**Velocity - Velocity**
X start	User input	User input
Y start	User input	User input* or calculated by EcoStruxure Machine Expert
Slope start	Fixed to 0	User input or calculated* by EcoStruxure Machine Expert
Curvature start	Fixed to 0	Fixed to 0
X end	User input	User input
Y end	User input	User input* or calculated by EcoStruxure Machine Expert
Slope end	Fixed to 0	The same value is used as for the parameter **Slope** start.
Curvature end	Fixed to 0	Fixed to 0
* By default, the **Slope** start parameter is calculated by EcoStruxure Machine Expert, whereas the parameters **Y** start and **Y** end are editable.You can change this by clicking the option next to the parameter.		

Parameters for Motion Law Sinus-straight line combination

The motion law **Sinus-straight line combination** is only available for **Start condition > Return** and **End condition > Return**. The table lists the available parameters:

Parameter	**Start condition - End condition**
	Return - Return
X start	User input
Y start	User input
Slope start	User input* or calculated by EcoStruxure Machine Expert
Curvature start	User input* or calculated by EcoStruxure Machine Expert
X end	User input
Y end	User input
Slope end	User input* or calculated by EcoStruxure Machine Expert
Curvature end	User input* or calculated by EcoStruxure Machine Expert
C	User input* or calculated by EcoStruxure Machine Expert
Lambda	User input* or calculated by EcoStruxure Machine Expert
* Use the **Input parameters** list to select two out of six parameters as editable. The remaining four parameters are calculated by EcoStruxure Machine Expert.	

Parameters for Motion Law General polynomial of the 5th degree

Since the motion law **General polynomial of the 5th degree** is available for 15 combinations of **Start condition** - **End condition** values, 4 different parameter tables are provided for this motion law.

The table lists the parameters that are available for the motion law **General polynomial of the 5th degree** and the values they can have in accordance with the selected **Start condition**=**Dwell** together with several **End condition** values:

Parameter	**Start condition - End condition**			
	Dwell - Dwell	**Dwell - Velocity**	**Dwell - Motion**	**Dwell - Return**
X start	User input	User input	User input	User input
Y start	User input	User input	User input	User input
Slope start	Fixed to 0	Fixed to 0	Fixed to 0	Fixed to 0
Curvature start	Fixed to 0	Fixed to 0	Fixed to 0	Fixed to 0
X end	User input	User input	User input	User input
Y end	User input	User input	User input	User input
Slope end	Fixed to 0	User input	User input	Fixed to 0
Curvature end	Fixed to 0	Fixed to 0	User input	User input

The table lists the parameters that are available for the motion law **General polynomial of the 5th degree** and the values they can have in accordance with the selected **Start condition**=**Motion** together with several **End condition** values:

Parameter	**Start condition - End condition**			
	Motion - Dwell	**Motion - Motion**	**Motion - Return**	**Motion - Velocity**
X start	User input	User input	User input	User input
Y start	User input	User input	User input	User input
Slope start	User input	User input	User input	User input
Curvature start	User input	User input	User input	User input
X end	User input	User input	User input	User input
Y end	User input	User input	User input	User input
Slope end	Fixed to 0	User input	Fixed to 0	User input
Curvature end	Fixed to 0	User input	User input	Fixed to 0

The table lists the parameters that are available for the motion law **General polynomial of the 5th degree** and the values they can have in accordance with the selected **Start condition**=**Return** together with several **End condition** values:

Parameter	**Start condition - End condition**			
	Return - Dwell	**Return - Motion**	**Return - Return**	**Return - Velocity**
X start	User input	User input	User input	User input
Y start	User input	User input	User input	User input
Slope start	Fixed to 0	Fixed to 0	Fixed to 0	Fixed to 0
Curvature start	User input	User input	User input	User input
X end	User input	User input	User input	User input
Y end	User input	User input	User input	User input
Slope end	Fixed to 0	User input	Fixed to 0	User input
Curvature end	Fixed to 0	User input	User input	Fixed to 0

The table lists the parameters that are available for the motion law **General polynomial of the 5th degree** and the values they can have in accordance with the selected **Start condition**=**Velocity** together with several **End condition** values:

Parameter	**Start condition - End condition**			
	Velocity - Dwell	**Velocity - Motion**	**Velocity - Return**	**Velocity - Velocity**
X start	User input	User input	User input	User input
Y start	User input	User input	User input	User input
Slope start	User input	User input	User input	User input
Curvature start	Fixed to 0	Fixed to 0	Fixed to 0	Fixed to 0
X end	User input	User input	User input	User input
Y end	User input	User input	User input	User input
Slope end	Fixed to 0	User input	Fixed to 0	User input
Curvature end	Fixed to 0	User input	User input	Fixed to 0

Parameters for Motion Law Modified acceleration trapezoid combination

The table lists the parameters that are available for the motion law **Modified acceleration trapezoid combination** and the values they can have in accordance with the selected **Start condition** and **End condition**:

Parameter	**Start condition - End condition**	
	Return - Dwell	**Dwell - Return**
X start	User input	User input
Y start	User input	User input
Slope start	Fixed to 0	Fixed to 0
Curvature start	User input	Fixed to 0
X end	User input	User input
Y end	User input	User input
Slope end	Fixed to 0	Fixed to 0
Curvature end	Fixed to 0	User input

Parameters for Motion Law Harmonic combination

The motion law **Harmonic combination** indicates the admissible range of values for velocity and acceleration to assist you in creating valid profiles.

> **NOTE:** As velocity and acceleration are interdependent, it may happen that borderline values are accepted but result in an invalid curve. In this case, the message *HarmComb: End curvature results in invalid profile 1* is displayed in the **Messages** view (see Menu Commands, Online Help).

The table lists the parameters that are available for the motion law **Harmonic combination** and the values they can have in accordance with the selected **Start condition** and **End condition**:

Parameter	**Start condition - End condition**			
	Return - Dwell	**Dwell - Return**	**Return - Velocity**	**Velocity - Return**
X start	User input	User input	User input	User input
Y start	User input	User input	User input	User input
Slope start	Fixed to 0	Fixed to 0	Fixed to 0	User input
Curvature start	User input	Fixed to 0	User input	Fixed to 0
X end	User input	User input	User input	User input
Y end	User input	User input	User input	User input
Slope end	Fixed to 0	Fixed to 0	User input	Fixed to 0
Curvature end	Fixed to 0	User input	Fixed to 0	User input

Generating IEC Source Code

Generated IEC Source Code Tab

The **Generated IEC Source Code** tab provides the program code that was generated automatically based on the cam diagram and the target specified in the **Configuration** tab.

If you define a target structure (cam data structure), the cam data is automatically written into this target structure during program initialization.

You can also configure a target structure manually by copying this code and replacing `###` by the name of your variable if no target has been defined.

The program code displayed in this tab is read-only. When the cam diagram is modified, the code is regenerated.

Configuration

Configuration Tab

The **Configuration** tab allows you to select a target structure for its generation and to define the properties.

The **Configuration** tab allows you to do the following:

- Select a target structure for source code generation.
- Define properties for this structure.

Element	Description
Targeted technology	The **Targeted technology** section provides two target structures (cam data structure) for source code generation: • **CMT.ST_MultiCam (Library: CommonMotionTypes)** • **EXCAM.IF_Cam (Library: ExtendedCam)** Cam data is initialized every time the application program is started. If the expression is undeterminable, a diagnostic message is generated. This function is not available for all controllers. Consult the *Programming Guide* specific to your controller for further information.
Code generation variant	The following options are available: • **Global variable instance (Variable name: '<YourCamDiagramName>')** • **Function block type including initialization code (Type name: '<YourCamDiagramName>')** • **Initialization function for SmartTemplate modules** • **Existing IEC structure**: Enter a structure or click the browse button (**...**) to open the **Input Assistant** dialog box to browse for a structure.
CMT.ST_MultiCam configuration	If **Targeted technology > CMT.ST_MultiCam (Library: CommonMotionTypes)** is selected, configure the values of the following parameters: • **User table Id**: Enter a positive integer (including 0). • **Y - Period**: Enter a positive floating-point number (including 0). The default value for both parameters is 0.

Script Hook Manager

What's in This Chapter

Script Hook Manager 546

Script Hook Manager

Overview

The **Script Hook Manager** provides an interface to several commands and events that can be executed while running EcoStruxure Machine Expert. The execution of the command or detection of the event can trigger the execution of a specific Python script. You can program your own Python script or you can use one of the Python templates that is delivered with EcoStruxure Machine Expert.

> **NOTE:** EcoStruxure Machine Expert supports different events. Events that are not supported by the programming software you are using are marked in the **Script Hook Manager** editor, page 547.

The *AutomaticProjectArchiveCreation.py* template, for example, provides a Python script for creating a project archive. If different users are working on the same project, this helps to ensure that a project archive is created whenever the project has been changed by whichever person. Thus, each user is supplied with the latest state of the project if the latest project archive is used.

For further information, refer to the following *Script Engine User Guides* provided in the *Software - Programming* part of the online help:

- CODESYS Script Engine - User Guide
- Schneider Electric Script Engine - User Guide

Creating a Script Hook Manager Object

To create a **Script Hook Manager** object, proceed as follows:

Step	Action	Comment
1	Right-click the **Global** node of the **Applications** tree and execute the command **Add Object > Script Hook Manager**.	**Result**: The **Add Script Hook Manager** dialog box opens.
2	Select the suitable option for creating a Python script.	• Select the **New Script** option to create a new Python script from scratch in Logic Builder. • Select the **From Template** option and select a Python template that is delivered with EcoStruxure Machine Expert as a basis for creating your Python script. The use cases that are covered by the different templates are described by the comments in the scripts themselves. • Select the **From Existing Script** option to open a Python script file that you programmed in another tool. Click the **Select** button to open a **File open** dialog box to browse for the *.py* file.
3	Click the **Add** button.	**Results**: • A **Script Hook Manager** node and a **Python Script** node are added as subnodes of the **Global** node of the **Applications tree**. • The **Script Hook Manager** editor opens.

Script Hook Manager Editor

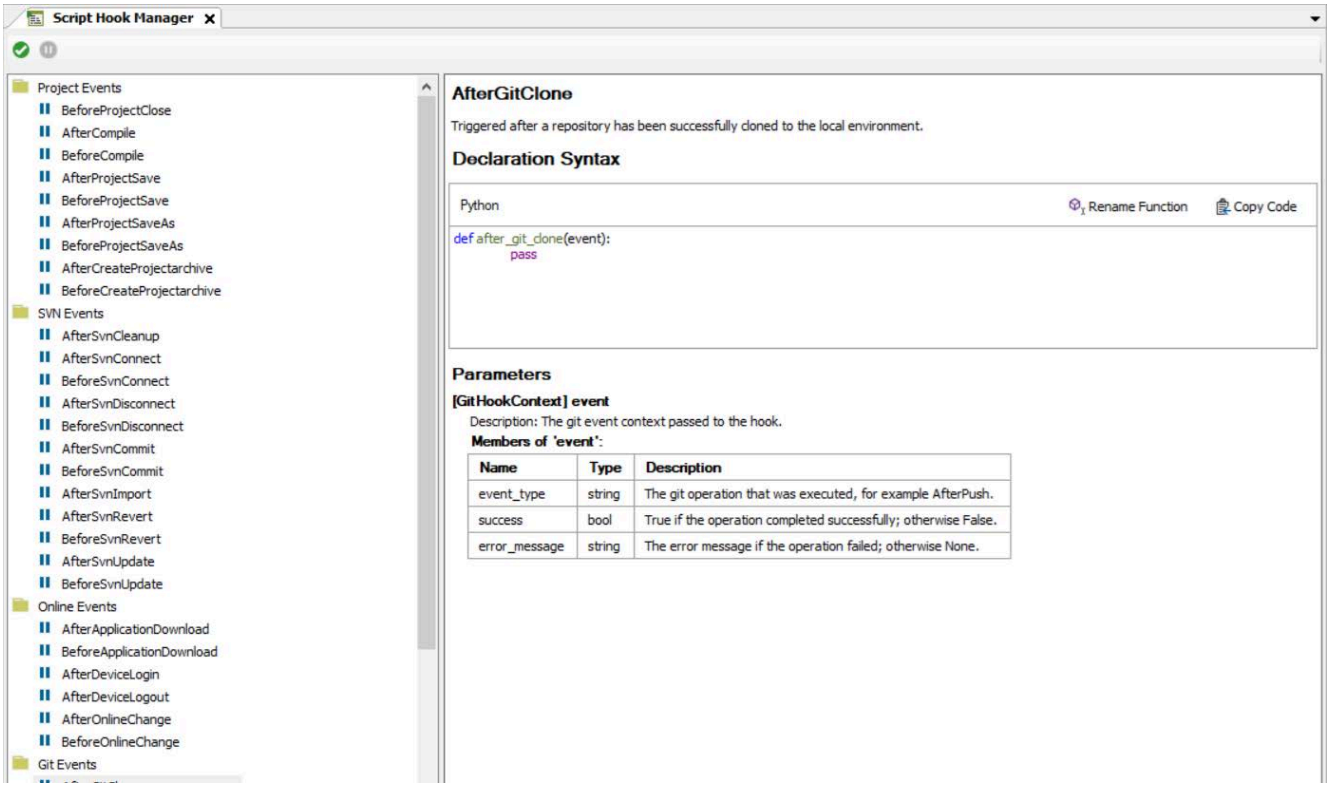
The **Script Hook Manager** editor lists the **Commands** and **Events** that are available for triggering the execution of Python scripts on the left-hand side.

- To activate a command or event, select the respective item, and click the **Enable Hook** button .
- To deactivate a command or event, select the respective item, and click the **Disable Hook** button .

These status settings are saved to the project. For example, if you open a project that was previously saved with a **BeforeProjectSave** hook enabled, this hook remains active and is triggered as soon as the event / command becomes valid.

NOTE: EcoStruxure Machine Expert supports different events. Events that are not supported by the programming software you are using are marked in the **Script Hook Manager** editor . When you attempt to execute an unsupported event, a message is displayed in the **Messages** view (see Menu Commands, Online Help) and the code will not be executed.

On the right-hand side, further information is presented for the selected command or event:



NOTE: Git Events are exclusive to file-based storage (*.fbs) projects.

The **Declaration Syntax** section provides the signature of the method that is called by the system after the event or command has been executed by EcoStruxure Machine Expert. Click the **Copy Code** button to insert the code into your Python code in the **Python Script** editor and extend this method by the Python commands you want to execute.

The **Rename Function** button allows you to modify the name of the Python function in a separate dialog box. After you have saved a different name, make sure that you use the exact string of the modified name in your Python code.

The **Parameters** section lists the parameters that are available inside the Python method when it is executed.

Python Script Editor

The **Python Script** editor allows you to program your Python script directly in Logic Builder. The editor supports all Python programming features, even though the Autocomplete (IntelliSense) function does not provide all available Python keywords for selection.

According to the selection you made in the **Add Script Hook Manager** dialog box, you have the following options for creating your Python script in the **Python Script** editor:

- From scratch, starting with an empty **Python Script** editor, by entering your code and using the code snippets you copy from the **Declaration Syntax** section of the **Script Hook Manager** editor.
- From template, by using the script provided by the selected template as a basis and extending it by your specific Python code.
- By importing the Python script you programmed in another tool.

  NOTE: After you have programmed your Python script, click the **Update Script Scope**  button to synchronize the script and to apply the modifications. If you exit the **Python Script** editor without synchronization, modifications are saved and are thus still available in the editor but they are not applied. The previous script without modification is still executed instead. If you load another project or create a new project, a different script - or no script at all - will be available.

With the **Update Script Scope** procedure, the Python code is validated. If errors are detected, they are indicated in the **Messages** view (see Menu Commands, Online Help) and the code will not be executed.

Ignoring Multiple Events During Login/Logout

NOTE: Due to technical limitations the *AfterDeviceLogin* and/or the *AfterDeviceLogout* event can be triggered several times.

For example, ignore the multiple execution of the *AfterDeviceLogin* event with the help of a flag in the script:

```
def after_device_login():
    if session_store.get("loginHandled", False) == False:
        print("after_device_login")
        session_store.set("loginHandled", True)
    pass

def after_device_logout():
    print("after_device_logout")
    session_store.set("loginHandled", False)
    pass
```

Script Hook Messages in the Messages View

The texts you program in your Python code with the `print` command are displayed in the **Scripthooks** area of the **Messages** view (see Menu Commands, Online Help).

Programming Reference

What’s in This Part

Variables Declaration 550
Data Types 635
Programming Guidelines 671
Operators 678
Operands 746

Variables Declaration

What's in This Chapter

Declaration 550
Variable Types 566
Method Types 579
Pragma Instructions 583
Attribute Pragmas 597
The Smart Coding Functionality 634

Declaration

General Information

Overview

You can declare variables:

- in the **Declaration Editor** of a POU, page 373
- via the **Auto Declare** dialog box, page 561
- in a GVL editor
- in the **PersistentVars** editor, page 218

The kind (in the tabular declaration editor it is named **Scope**) of the variables to be declared is specified by the keywords embracing the declaration of one or several variables. In the textual declaration editor, page 373, the common variable declaration is embraced by `VAR` and `END_VAR`.

For further variable declaration scopes, refer to:

- `VAR_INPUT`
- `VAR_OUTPUT`
- `VAR_IN_OUT`
- `VAR_GLOBAL`
- `VAR_TEMP`
- `VAR_STAT`
- `VAR_INST`
- `VAR_EXTERNAL`
- `VAR_CONFIG`

The variable type keywords may be supplemented by attribute keywords, page 570.

Example: `RETAIN (VAR_INPUT RETAIN)`

Syntax

Syntax for variable declaration:

<Identifier> {`AT` <address>}:<data type> {:=<initialization>};

The parts in braces {} are optional.

Identifier

The identifier is the name of a variable.

Consider the following facts when defining an identifier.

- no spaces or special characters allowed
- no case-sensitivity: `VAR1`, `Var1` and `var1` are all the same variable
- recognizing the underscore character: `A_BCD` and `AB_CD` are considered two different identifiers. Do not use more than one underscore character in a row.
- unlimited length
- no keyword name allowed, such as `VAR_GLOBAL`
- multiple use (see next paragraph)

Also, consider the chapter Best Practices for the Naming of Identifiers, page 553.

Multiple Use of Identifiers (Namespaces)

The following outlines the regulations concerning the multiple use of identifiers:

- Duplicate use of identifiers is not allowed locally.
- Multiple use of an identifier is allowed globally: a local variable can have the same name as a global one. In this case, the local variable within the POU will have priority.
- A variable defined in a global variable list (GVL) can have the same name as a variable defined in another global variable list (GVL). In this context, consider the following IEC 61131-3 extending features:
  - Global scope operator: an instance path starting with a dot (`.`) opens a global scope. So, if there is a local variable, for example `ivar`, with the same name as a global variable, `.ivar` refers to the global variable.
  - You can use the name of a global variable list (GVL) as a namespace for the included variables. You can declare variables with the same name in different global variable lists (GVL). They can be accessed specifically by preceding the variable name with the list name.

    Example

    ```
    globlist1.ivar := globlist2.ivar;
    ```

    (* *ivar* from *globlist2* is copied to *ivar* in GVL *globlist1* *)
  - Variables defined in a global variable list of an included library can be accessed according to syntax <library namespace>.<name of GVL>.<variable>.

    Example:

    ```
    globlist1.ivar := lib1.globlist1.ivar
    ```

    (* *ivar* from *globlist1* in library *lib1* is copied to *ivar* in GVL *globlist1* *)
- For a library also, a namespace is defined when it gets included via the **Library Manager**. So you can access a library module or variable by <library namespace>.<modulename|variablename>. Consider that, in case of nested libraries, the namespaces of all libraries concerned have to be stated successively.

  Example: If `Lib1` is referenced by `Lib0`, the module `fun` being part of `Lib1` is accessed by `Lib0.Lib1.fun`:

  `ivar := Lib0.Lib1.fun(4, 5);` (* return value of *fun* is copied to variable *ivar* in the project *)

  > **NOTE:** Once the checkbox **Publish all IEC symbols to that project as if this reference would have been included there directly.** has been activated within the **Properties** dialog box of the referenced library `Lib`, the module `fun` may also be accessed directly via `Lib0.fun`.
- Variables declared in GVLs or POUs of the **Global** node of the **Applications tree** can be accessed by prefixing them with the operator `"__POOL."`.

Backtick Identifiers

To use characters in identifiers that are usually not permitted, such as special characters, you can use backticks.

The grave accent character is used for backticks: ` (Unicode: U+0060)

It is a good practice to use backticks to be able to use the same identifier as in other programming languages or documents, such as circuit diagrams.

Any character can be used between two backticks, except line breaks and other backtick characters. Keywords are also allowed as identifiers between backticks.

Examples

```
`Variable+9`
`INT`
```

The backtick characters are part of each identifier. Thus, `var1` and `` `var1` `` are two different identifiers.

```
PROGRAM PLC_PRG
VAR
    var1 : INT;
    `var1`: INT;
END_VAR
var1 := var1 + 1;
`var1` := 12;
```

AT <address>

You can link the variable directly to a definite address, page 562 using the keyword `AT`.

In function blocks, you can also specify variables with incomplete address statements. In order that such a variable can be used in a local instance, an entry has to exist for it in the variable configuration.

Type

Valid data type, page 635, optionally extended by an :=< initialization>, page 560.

Pragma Instructions

Optionally, you can add pragma instructions, page 583 in the declaration part of an object in order to affect the code generation for various purposes.

Hints

Automatic declaration, page 561 of variables is also possible.

For faster input of the declarations, use the shortcut mode, page 561.

Best Practices for the Naming of Identifiers

Overview

Identifiers are defined:

- at the declaration of variables (variable name)
- at the declaration of user-defined data types
- at the creation of POUs (functions, function blocks, programs)

In addition to the general items to be considered when defining an identifier (refer to chapter *General Information* on variables declaration, page 550), consider the following in order to make the naming as unique as possible:

- Variable names, page 553
- Variable names in Libraries, page 556
- User-defined data types (DUTs) in Libraries , page 556
- Functions, Function blocks, Programs (POU), Actions , page 556
- POUs in Libraries , page 557

NOTE: The naming conventions and examples provided in the following paragraphs are based on good practices applied throughout the development of Schneider Electric libraries and correspond to the default convention queries that are used within EcoStruxure Machine Expert Code Analysis. Alternative naming conventions from other organizations are provided in parentheses in the following sections.

Variable Names

For naming variables in applications and libraries, follow the Hungarian notation as far as possible.

Find for each variable a meaningful, short description. This is used as the base name. Use a capital letter for each word of the base name. Use small letters for the rest (example: `FileSize`).

Data Type	Lower Limit	Upper Limit	Information Content	Prefix	Comment
BOOL	FALSE	TRUE	1 bit	x*	–
				b	reserved
BYTE	–	–	8 bit	by	bit string, not for arithmetic operations
WORD	–	–	16 bit	w	bit string, not for arithmetic operations
DWORD	–	–	32 bit	dw	bit string, not for arithmetic operations
LWORD	–	–	64 bit	lw	bit string, not for arithmetic operations
SINT	–128	127	8 bit	si	arithmetic integer data type, 8-bit
USINT	0	255	8 bit	usi	arithmetic integer data type, 8-bit
INT	–32,768	32,767	16 bit	i	arithmetic integer data type, 16-bit
UINT	0	65,535	16 bit	ui	arithmetic integer data type, 16-bit
DINT	–2,147,483,648	2,147,483,647	32 bit	di	arithmetic integer data type, 32-bit
UDINT	0	4,294,967,295	32 bit	udi	arithmetic integer data type, 32-bit
LINT	-2^{63}	$2^{63}-1$	64 bit	li	arithmetic integer data type, 64-bit
ULINT	0	$2^{64}-1$	64 bit	uli	arithmetic integer data type, 64-bit
REAL	–	–	32 bit	r	arithmetic floating-point data type, 32-bit
LREAL	–	–	64 bit	lr	arithmetic floating-point data type, 64-bit
STRING	–	–	–	s	Single-byte character string of variable length (default setting: 80 characters)
WSTRING	–	–	–	ws	Double-byte character string of variable length (default setting: 80 characters)
TIME	–	–	–	tim	time duration, 32-bit
LTIME	–	–	–	ltim	time duration, 64-bit
• TIME_OF_DAY • TOD	–	–	–	tod	time of day, 32-bit
• LTIME_OF_DAY • LTOD	–	–	–	ltod	time of day, 64-bit
• DATE_AND_ TIME • DT	–	–	–	dt	date and time
• LDATE_AND_ TIME • LDT	–	–	–	ldt	–
DATE	–	–	–	• dat • d	calendar date
LDATE	–	–	–	• ldat • ld	calendar date

Data Type	Lower Limit	Upper Limit	Information Content	Prefix	Comment
Enumeration	–	–	–	et (e)	–
POINTER	–	–	–	p	–
ARRAY	–	–	–	a	–
Structure	–	–	–	st	–
Function block	–	–	–	fb	–
Interface	–	–	–	if (itf)	–
Union	–	–	–	ut	–
* intentionally for boolean variables x is chosen as a prefix in order to differentiate from BYTE and also in order to accommodate the perception of an IEC programmer (see addressing %IX0.0).					

Simple declaration

Examples for simple declarations:

```
bySubIndex: BYTE;
```

```
sFileName: STRING;
```

```
udiCounter: UDINT;
```

Nested declaration

Example for a nested declaration where the prefixes are attached to each other in the order of the declarations:

```
pabyTelegramData: POINTER TO ARRAY [0..7] OF BYTE;
```

Function block instances and variables of user-defined data types

Function block instances and variables of user-defined data types get a shortcut for the function block or the data type name as a prefix (for example: stSDO).

Example

```
stSDOReceivedTelegram: ST_SDOTelegram;
TYPE ST_SDOTelegram :
STRUCT
  wIndex:WORD;
  bySubIndex:BYTE;
  byLen:BYTE;
  aby: ARRAY [0..3] OF BYTE;
END_STRUCT
END_TYPE
```

Local constants

Local constants (c) start with prefix c and an attached underscore, followed by the type prefix and the variable name.

Example

```
VAR CONSTANT
  c_uiSyncID: UINT := 16#80;
END_VAR
```

Global variables and global constants

Global variables are prefixed by G_ (g_) and global constants are prefixed by Gc_ (gc_).

Example

```
VAR_GLOBAL
  G_iTest: INT;
END_VAR
VAR_GLOBAL CONSTANT
  Gc_dwExample: DWORD;
END_VAR
```

Variable Names in Libraries

Structure

Basically, refer to the above description for variable names. Use the library namespace as prefix, when accessing a variable in your application code.

Example

```
G_iTest: INT; (declaration)
CAN.G_iTest (implementation, call in an application program
```

User-Defined Data Types (DUT) in Libraries

Structure

The name of each structure data type consists of the corresponding type prefix (also refer to POU Prefix, page 676) and a short, meaningful name (for example, `ET_Day`) of the structure.

Example (in library with namespace `CAL`):

```
TYPE ET_Day :(
MONDAY,
TUESDAY,
WEDNESDAY,
THURSDAY,
FRIDAY,
SATURDAY,
SUNDAY);
```

Declaration:

```
etToday: CAL.ET_Day;
```

Use in application:

```
IF etToday = CAL.ET_Day.MONDAY THEN
```

> **NOTE:** Consider the usage of the namespace when using DUTs or enumerations declared in libraries.

Functions, Function Blocks, Programs (POU), Actions

The names of functions, function blocks, and programs consist of the corresponding type prefix (also refer to POU Prefix, page 676) and a short, meaningful name of the POU (for example, `FB_SendTelegram`). As with variables, the first letter of a word of the POU name should be an uppercase letter whereas the others should be lowercase letters. For example, compose the name of the POU of a verb and a substantive to clearly indicate the operation of the function block.

Example

```
FUNCTION_BLOCK FB_SendTelegram
```

In the declaration part, provide a short description of the POU as a comment. Further on, the inputs and outputs should be provided with comments. In case of function blocks, insert the associated prefix for set-up instances directly after the name.

Actions

Actions do not get a prefix. Only those actions that are to be called only internally that is by the POU itself, start with `prv_`.

POUs in Libraries

Structure

For creating method names, the same rules apply as for actions. Enter comments for possible inputs of a method. Add a short description of a method to its declaration. Start interface names with prefix `IF_` (`I`); for example, `IF_CANDevice`.

NOTE: Consider the usage of the namespace when using POUs declared in libraries.

Shadowing Rules

Overview

In general, there are no restrictions in EcoStruxure Machine Expert that prevent you from using the same identifier for different elements. You can, for example, assign the same name to a local function block instance and to a function.

In the following “bad practice” example, it is unclear whether the instance or the function is called in the program:

```
FUNCTION YYY : INT
;
END_FUNCTION

FUNCTION_BLOCK XXX
;
END_FUNCTION_BLOCK

PROGRAM PLC_PRG
VAR
    YYY : XXX;
END_VAR
YYY();
END_PROGRAM
```

To help prevent creating ambiguous code, make frequent use of the following good practices:

- Follow naming conventions, for example, by using certain prefixes for variables. For further information, refer to the *Best Practices for the Naming of Identifiers*, page 553.
- Use the Code Analysis component provided with EcoStruxure Machine Expert for verifying the naming of variables.
- Make consistent use of the *qualified_only* attribute, page 629 for enumerations and global variable lists.
- Use the library namespace prefix, page 676 to access elements of the library.
- Use the prefix `__POOL` to explicitly call a POU from the **Global** node of the **Applications tree** and to avoid calling a POU of the same name from the **Devices tree**.

  Example: `svar_pou := __POOL.POU();`

Compiler Shadowing Multiple Declarations

The EcoStruxure Machine Expert compiler does not report errors if the same identifier is used for different elements but it searches the code for the declaration of the identifier in a specific order. As soon as a declaration is found, the search is stopped. Other declarations that might exist in the code cannot be detected, they are “shadowed” for the compiler. The following paragraph describes the shadowing rules of the compiler, i.e. the order that is applied when searching for declarations of identifiers. The paragraph *Ambiguous access and qualified*

access, page 559 provides ways to help prevent ambiguous access and to bypass shadowing rules.

Search Order in the Application

When the compiler detects a single identifier in the code of an application, it searches for the corresponding declaration in the following order:

Stage	Sub-stage	Description
1		Local variables
	a	Local variables of a method.
	b	Local variables in the function block, program, or function, and in base function blocks.
	c	Local methods of the POU.
2		Global variables
	a	Global variables in the application if the `qualified_only` attribute is not set in the variable list where the global variables are declared.
	b	Global variables in a parent application if the `qualified_only` attribute is not set in the variable list where the global variables are declared.
	c	Global variables in referenced libraries when neither the library nor the variable list requires qualified access.
3		POU or type names
	a	POU or type names from the application (such as names of global variable lists, function blocks).
	b	POU or type names from a parent application.
	c	POU or type names from a library.
4		Libraries
	a	Namespaces of locally referenced libraries and libraries that are published by libraries.
5		Global variables in the **Applications tree**
	a	Global variables in the **Applications tree** if the `qualified_only` attribute is not set in the variable list where they are declared.
	b	POU or type names from the **Applications tree** (such as names of global variable lists, function blocks).
	c	Libraries from the **Applications tree**.

NOTE: Libraries that are inserted in the **Library Manager** that is a subnode of the **Global** node of the **Tools tree** are available in all applications in the project with the placeholder resolution. These libraries form a common namespace with the libraries in the application.

As described in the section Description of the **Tools Tree**, page 42, the **Library Manager** is available at two different locations in the **Tools tree**: as subnode of the **Global** node and as subnode of the **Application** node.

NOTE: In case the same library was added to both **Library Manager** nodes, the symbols of the library below the **Application** node are used.

Search Order in the Library

When the compiler detects a single identifier in the code of a library, it searches for the corresponding declaration in the following order:

Stage	Description
1	Local variables of a method.
2	Local variables in the function block, program, or function, and in base function blocks.
3	Local methods of the POU.
4	Global variables in the local library if the *qualified_only* attribute is not set in the variable list where the global variables are declared.
5	Global variables in referenced libraries when neither the library nor the variable list requires qualified access.
6	POU or type names from the local library (such as names of global variable lists, function blocks).
7	POU or type names from a referenced library.
8	Namespaces of locally referenced libraries and libraries that are published by locally referenced libraries.

Ambiguous Access and Qualified Access

These search orders cannot completely eliminate ambiguous access. Ambiguous access still exists, for example, if a variable with the same name exists in two global variable lists that do not require qualified access. If such a case is detected, the compiler reports the error: `ambiguous use of the name XXX`. To help avoid this, use qualified access, for example by adding the name of the global variable list (example: `GVL.XXX`).

Use qualified access to help avoid shadowing:

- Use the name of the global variable list to provide unique access to a variable in the list.
- Use the name of a library to provide unique access to elements in the library.
- Use the *THIS* pointer to provide unique access to variables in a function block even if a local variable with the same name exists in a method of the function block.

To find the declaration location of an identifier, execute the **Edit > Browse > Go to Definition** command (see Menu Commands, Online Help).

Searching in Instance Paths

The search orders described in the previous paragraphs do not apply to:

- Identifiers that exist as components of an instance path.
- Identifiers that are used as inputs in calls.

When attempting to access `yy.component`, the entity `yy` determines the location that is searched for detecting the declaration of `component`.

If...	Then ...
If `yy` denotes a variable with a structured data type (STRUCT or UNION),	Then `component` is searched in the following order: 1. Local variables of the function block 2. Local variables of the base function block 3. Methods of the function block 4. Methods of the base function block
If `yy` denotes a global variable list or a program,	Then `component` is searched in this list only.
If `yy` denotes a namespace of a library,	Then `component` is searched in this library using the order as described in the paragraph *Search Order in the Library*, page 559.

In the second instance, the compiler evaluates whether access to the element is allowed (the variable may only be locally accessible, or a method may be private). If access is not allowed, an error message is generated.

Variables Initialization

Default Initialization Value

The default initialization value is 0 for all declarations, but you can add user-defined initialization values in the declaration of each variable and data type.

User-Defined Initialization Values

The user-defined initialization is brought about by the assignment operator `:=` and can be any valid ST expression. Thus, constant values as well as other variables or functions can be used to define the initialization value. Verify that a variable used for the initialization of another variable is already initialized itself.

Example of valid variable initializations:

```
VAR
var1:INT := 12;                 * Integer variable with
initial value of 12. *
x : INT := 13 + 8;              * Integer value defined an
expression with literal values.*
y : INT := x + fun(4);          * Integer value defined by an
expression containing a function call. NOTE: Be sure that
any variables used in the variable initialization have
already been defined. *
z : POINTER TO INT := ADR(y); * POINTER is not described by
the IEC61131-3: Integer value defined by an address
function; NOTE: The pointer will not be initialized if the
declaration is modified online. *
END_VAR
```

Further Information

For further information, refer to the following descriptions:

- initializing arrays, page 651

- initialization of structures, page 661
- initialization of a variable with a subrange type, page 666

NOTE: Variables of global variables lists (GVL) are initialized before local variables of a POU.

NOTE: As from SoMachine version 4.0, variables in a function block are initialized in the following order: First, the constants in accordance with the order of their declarations, then the other variables in accordance with the order of their declarations.

For further information regarding initialization order, refer to the Attribute global_init_slot, page 607.

Declaration

Declaration Types

You can declare variables manually by using the textual or tabular declaration editor, page 373 or automatically like explained in this chapter.

Automatic Autodeclaration

You can define in the **Options** dialog box, category **Text editor > Editing**, that the **Auto Declare** dialog box should open as soon as a not yet declared string is entered in the implementation part of an editor and the ENTER key is pressed. This dialog box supports the declaration of the variable, page 550.

Manual Autodeclaration

To open the **Auto Declare** dialog box manually:

- execute the command **Auto Declare**, which by default is available in the **Edit** menu or
- press the keys SHIFT+F2

If you select an already declared variable before opening the **Auto Declare** dialog box (see Menu Commands, Online Help), you can edit the declaration of this variable.

Shortcut Mode

Overview

The declaration editor, page 373 and the other text editors where declarations are performed support the shortcut mode.

Activate this mode by pressing CTRL+ENTER when you end a line of declaration.

It allows you to use shortcuts instead of completely typing the declaration.

Supported Shortcuts

The following shortcuts are supported:

- All identifiers up to the last identifier of a line will become declaration variable identifiers.
- The type of declaration is determined by the last identifier of the line.

In this context, the following replacements are performed:

B or BOOL	is replaced by	BOOL
I or INT		INT
R or REAL		REAL
S or string		STRING

- If no type has been established through these rules, automatically BOOL is the type and the last identifier will not be used as a type (see example 1).
- Every constant, depending on the type of declaration, will turn into an initialization or a string (see examples 2 and 3).
- An address (as in %MD12) is extended by the AT keyword (see example 4).
- A text after a semicolon (;) becomes a comment (see example 4).
- All other characters in the line are ignored (see, for example, the exclamation point in example 5).

Examples

Example No.	Shortcut	Resulting Declaration
1	A	A: BOOL;
2	A B I 2	A, B: INT := 2;
3	sX S 2; A string	sX:STRING(2); // A string
4	X %MD12 R 5; Real Number	X %MD12 R 5 Real Number
5	B !	B: BOOL;

AT Declaration

Overview

In order to link a project variable with a definite address, you can assign variables to an address in the **I/O Mapping** view of a device in the controller configuration (device editor). Alternatively you can enter this address directly in the declaration of the variable.

Syntax

```
<variable name> AT <address> : <data type>;
<address> : %<memory area prefix> ( <size prefix> )? <memory
position
```

A valid address has to follow the keyword `AT`. For further information, refer to the *Address* description, page 760. Consider possible overlaps in case of byte addressing mode.

This declaration allows assigning a meaningful name to an address. Any changes concerning an incoming or outgoing signal may only be done in a single place (for example, in the declaration).

Consider the following when choosing a variable to be assigned to an address:

- Variables requiring an input cannot be accessed by writing. The compiler intercepts this detecting an error.
- `AT` declarations only can be used with local or global variables. They cannot be used with input and output variables of POUs.
- `AT` declarations are not allowed in persistent variable lists.
- If `AT` declarations are used with structure or function block members, all instances will access the same memory location of this structure / function block. This corresponds to static variables in classic programming languages such as C.
- The memory layout of structures is determined by the target as well.

Examples

```
xCounterHeat7 AT %QX0.0: BOOL;
xLightCabinetImpulse AT %IX7.2: BOOL;
xDownload AT %MX2.2: BOOL;
```

Note

Individual bit addresses that are mapped to Boolean variables have restrictions. It is not possible to use these variables with `VAR_IN_OUT`, `REFERENCE TO`, or the `ADR` operator.

If boolean variables are assigned to a BYTE, WORD or DWORD address, they occupy 1 byte with TRUE or FALSE, not just the first bit after the offset.

Example:

```
PROGRAM PLC_PRG
VAR
    xVar AT %QW0 : BOOL;
END_VAR
xVar := TRUE;
```

In this example, when the variable `xVar` is written, the output memory range `QX0.0...QX0.7` is affected.

The memory size for input, output, and memory data (declarations with `AT %I`, `%Q` and `%M`) is predefined by the target device and can be overwritten in the properties of an application object (see Menu Commands, Online Help) for PacDrive controllers (PacDrive LMC Eco, PacDrive LMC Pro/Pro2).

Keywords

Overview

Write keywords in uppercase letters in the editors.

The following strings are reserved as keywords. They cannot be used as identifiers for variables or POUs:

- _
- &
- (
-)
- *
- ,
- .
- ..
- /
- :
- :=
- ;
- [
-]
- ^
- __CATCH
- __CHECKLICENSE
- __CHECKLICENSEBIT
- __DELETE
- __ENDTRY
- __FINALLY
- __ISVALIDREF
- __MEMORYBARRIER
- __NEW
- __QUERYINTERFACE
- __QUERYPOINTER
- __THROW
- __TRY
- __XADO
- |
- +
- <
- <=
- <>
- =
- >=
- >
- =>
- ABS
- ACOS
- ADR
- AND
- AND_THEN
- ASIN
- ATAN

- BITADR
- BY
- CASE
- CONTINUE
- COS
- DO
- ELSE
- ELSIF
- END_CASE
- END_FOR
- END_IF
- END_REPEAT
- END_WHILE
- EXIT
- EXP
- EXPT
- FALSE
- FOR
- IF
- INI
- LIMIT
- LN
- LOG
- LOWER_BOUND
- MAX
- MIN
- MOD
- MOVE
- MUX
- NOT
- OR
- OR_ELSE
- R=
- REF=
- REPEAT
- RETURN
- ROL
- ROR
- S=
- SEL
- SHL
- SHR
- SIN
- SIZEOF
- SQRT
- SUPER
- TAN

- THEN
- THIS
- TO
- TRUE
- TRUNC
- TRUNC_INT
- UNTIL
- UPPER_BOUND
- WHILE
- XOR

Additionally, the conversion operators as listed in the **Input Assistant** are handled as keywords.

Variable Types

Variable Types

Overview

This chapter provides further information on the following variable types:

- VAR local variables, page 566
- VAR_INPUT input variables, page 567
- VAR_OUTPUT output variables, page 567
- VAR_IN_OUT input and output variables, page 567
- VAR_IN_OUT CONSTANT transfer variable, page 568
- VAR_GLOBAL global variables, page 569
- VAR_TEMP temporary variables, page 569
- VAR_STAT static variables, page 569
- VAR_EXTERNAL external variables, page 570
- VAR_INST instance variables, page 570

Local Variables - VAR

Between the keywords VAR and END_VAR, all local variables of a POU are declared, page 550. These have no external connection; in other words, they cannot be written from the outside.

Consider the possibility of adding an attribute, page 570 to VAR.

Example

```
VAR
 iLoc1:INT; (* 1. Local Variable*)
END_VAR
```

Input Variables - VAR_INPUT

Between the keywords VAR_INPUT and END_VAR, all variables are declared, page 550 that serve as input variables for a POU. Therefore, at the call position, the value of the variables can be provided along with a call.

Consider the possibility of adding an attribute, page 570.

Example

```
VAR_INPUT
 iIn1:INT (* 1. Inputvariable*)
END_VAR
```

Output Variables - VAR_OUTPUT

Between the keywords VAR_OUTPUT and END_VAR, all variables are declared that serve as output variables of a POU. Therefore, these values are carried back to the POU that makes the call.

Consider the possibility of adding an attribute, page 570 to VAR_OUTPUT.

Example

```
VAR_OUTPUT
 iOut1:INT; (* 1. Outputvariable*)
END_VAR
```

Output variables in functions and methods:

According to IEC 61131-3 draft 2, functions (and methods) can have additional outputs.You can assign them in the call of the function as shown in the following example.

Example

```
fun(iIn1 := 1, iIn2 := 2, iOut1 => iLoc1, iOut2 => iLoc2);
```

Input and Output Variables - VAR_IN_OUT

Between the keywords VAR_IN_OUT and END_VAR, all variables are declared, page 550 that serve as input and output variables for a POU.

NOTE: With variables of IN_OUT type, the value of the transferred variable is changed (transferred as a pointer, Call-by-Reference). Therefore, the input value for such variables cannot be a constant. For this reason, even the VAR_IN_OUT variables of a function block cannot be read or written directly from outside using <FBinstance>.<InOutVariable>.

NOTE: Do not assign bit-type symbols (such as *%MXaa.b* or BOOL variables that are located on such a bit-type address) to BOOL-type VAR_IN_OUT parameters of function blocks. If any such assignments are detected, they are reported as a detected **Build** error in the **Messages** view (see Menu Commands, Online Help).

Example

```
VAR_IN_OUT
 iInOut1:INT; (* 1. Inputoutputvariable *)
END_VAR
```

Transfer Variable - VAR_IN_OUT CONSTANT

A VAR_IN_OUT CONSTANT variable serves as a constant pass-by-reference parameter. A variable or a constant (literal) of type STRING or WSTRING can be passed to this parameter. The parameter is read-only. Passing of properties is not permitted.

Syntax

```
VAR_IN_OUT CONSTANT
    <variable name> : <data type>; // formal parameter
END_VAR
```

VAR_IN_OUT CONSTANT variables are declared without assigning an initialization value.

Usage

- When calling the POU, a constant variable or literal of type STRING or WSTRING can be passed. Write access is not permitted.
- When passing parameters of a STRING constant, the length of string is not restricted. The string length does not depend on the string length of the VAR_IN_OUT CONSTANT variables.

  NOTE: If the **Replace constants** option is selected in the **Project Settings > Compile options** dialog box (see Menu Commands, Online Help), the compiler generates an error message when parameters of a constant with elementary data type, page 635 or a constant variable with elementary data type are passed.

The variable is supported by EcoStruxure Machine Expert V2.0 and later versions.

Example: Passing parameters of STRING constants and STRING variables

```
FUNCTION funManipulate : BOOL
VAR_IN_OUT
    sReadWrite : STRING(16); (* Can be read or written here
in POU *)
    dwVarReadWrite : DWORD; (* Can be read or written here
in POU *)
END_VAR
VAR_IN_OUT CONSTANT
    c_sReadOnly : STRING(16); (* Constant string variable
can only be read here in POU *)
END_VAR
sReadWrite := 'String_from_POU';
dwVarReadWrite := STRING_TO_DWORD(c_sReadOnly);
PROGRAM PRG_A
VAR
    sVarFits : STRING(16);
    sValFits : STRING(16) :='1234567890123456';
    dwVar: DWORD;
END_VAR
// The following line of code causes the compiler error
C0417:
// C0417: VAR_IN_OUT parameter 'sReadWrite' needs a variable
with write access as input.
funManipulate(sReadWrite:='1234567890123456', c_sReadOnly:
='1234567890123456', dwVarReadWrite := dwVar);
// Correct code
funManipulate(sReadWrite := sValFits, c_sReadOnly := '23',
dwVarReadWrite := dwVar);
funManipulate(sReadWrite := sVarFits, c_sReadOnly :=
sValFits, dwVarReadWrite :=dwVar);
```

In the code example, strings are passed to the *funManipulate* function using different VAR_IN_OUT variables:

- When passing a string literal to a VAR_IN_OUT variable, a compiler error is issued.
- When passing a constant variable to a VAR_IN_OUT CONSTANT variable, correct code is generated. This even applies for passing string variables.

Global Variables - VAR_GLOBAL

You can declare normal variables, constants, external, or remanent variables that are known throughout the project as global variables. To declare global variables, use the global variable lists (GVL). You can add a GVL by executing the **Add Object** command (by default in the **Project** menu).

Declare the variables locally between the keywords VAR_GLOBAL and END_VAR.

Consider the possibility of adding an attribute, page 570 to VAR_GLOBAL.

A variable is recognized as a global variable by a preceding dot, for example, .iGlobVar1.

For detailed information on multiple use of variable names, the global scope operator dot (.) and name spaces refer to the chapter *Global Scope Operator*, page 739.

Global variables can only be declared in global variable lists (GVLs). They serve to manage global variables within a project. You can add a GVL by executing the **Add Object** command (by default in the **Project** menu).

NOTE: A variable defined locally in a POU with the same name as a global variable will have priority within the POU.

NOTE: Global variables are initialized before local variables of POUs.

Temporary Variables - VAR_TEMP

This feature is an extension to the IEC 61131-3 standard.

Temporary variables get (re)initialized at each call of the POU. VAR_TEMP declarations are only possible within programs and function blocks. These variables are also only accessible within the body of the program POU or function block.

Declare the variables locally between the keywords VAR_TEMP and END_VAR.

NOTE: You can use VAR_TEMP instead of VAR to reduce the memory space needed by a POU (for example inside a function block if the variable is only used temporarily).

Static Variables - VAR_STAT

This feature is an extension to the IEC 61131-3 standard.

Static variables can be used in function blocks, methods, and functions. Declare them locally between the keywords VAR_STAT and END_VAR. They are initialized on download.

Such as global variables, static variables do not lose their value after the POU in which they are declared is left. They are shared between the POUs they are declared in (for example, several function block instances, functions or methods share the same static variable). They can be used, for example, in a function as a counter for the number of function calls.

Consider the possibility of adding an attribute, page 570 to VAR_STAT.

External Variables - VAR_EXTERNAL

These are global variables which are imported into the POU.

Declare them locally between the keywords VAR_EXTERNAL and END_VAR and in the global variable list (GVL). The declaration and the global declaration have to be identical. If the global variable does not exist, a message will display.

NOTE: It is not necessary to define variables as external. These keywords are provided in order to maintain compatibility to IEC 61131-3.

Example

```
FUNCTION_BLOCK FB_DoSomething
VAR_EXTERNAL
 iVarExt1:INT; (* 1st external variable *)
END_VAR
```

Instance Variables - VAR_INST

If you declare a variable of a method as an instance variable by using the VAR_INST attribute, then this variable is not stored on the method stack but on the stack of the function block instance. It therefore behaves like other variables of the function block instance and is not reinitialized when the method is called.

VAR_INST variables are only allowed in methods. You can access such variables within the method only. Attributes such as CONST, RETAIN are not allowed in the declaration. The values of the variables can be monitored in the declaration part of the method.

Example

```
METHOD meth_last : INT
VAR_INPUT
  iVar : INT;
END_VAR
VAR_INST
  iLast : INT := 0;
END_VAR
meth_last := iLast;
iLast := iVar;
```

Attribute Keywords for Variable Types

Overview

You can add the following attribute keywords to the declaration, page 550 of the variable type in order to specify the scope:

- RETAIN: refer to *Retain Variables*, page 572
- PERSISTENT: refer to *Persistent Variables*, page 573
- CONSTANT: refer to *Constants - CONSTANT*, page 574, *Typed Literals*, page 577
- VAR_GENERIC CONSTANT: refer to Constants - VAR_GENERIC CONSTANT, page 575

Remanent Variables - RETAIN, PERSISTENT

Remanent variables can retain their value throughout the usual program run period. Declare them as retain variables or even more stringent as persistent variables.

The declaration determines the degree of resistance of a remanent variable in the case of resets, downloads, or a reboot of the controller. In applications mainly the combination of both remanent flags is used (refer to *Persistent Variables*, page 573).

NOTE: A `VAR PERSISTENT` declaration is interpreted in the same way as a `VAR PERSISTENT RETAIN` or `VAR RETAIN PERSISTENT`.

NOTE: Use the **Add all instance paths** command (see Menu Commands, Online Help) to take variables declared as persistent into the **Persistent list** object.

Persistence is achieved by the following procedures:

- Determining the cyclic tasks that access the variable.
- Copying the variable to the persistent global variable list at the end of the first cyclic task (in each cycle).
- Copying the value of the persistent variable to the normal variable at controller restart.

As memory is allocated both at the declaration and at the instance path, this persistent variable requires double memory allocation in each cycle. This can lead to extended processing times, especially with large, structured values.

Retain Variables

Variables declared as retain variables are stored in a nonvolatile memory area. To declare this kind of variable, use the keyword `RETAIN` in the declaration part of a POU or in a global variable list.

Example

```
VAR RETAIN
 iRem1 : INT; (* 1. Retain variable*)
END_VAR
```

Retain variables maintain their value even after an unanticipated shutdown of the controller as well as after a normal power cycle of the controller (or when executing the **Online** command **Reset Warm**). At restart of the program, the retained values will be processed further on. The other (non-retain) variables are newly initialized, either with their initialization values or with their default initialization values (in case no initialization value was declared).

For example, you may want to use a retained value when an operation, such as piece counting in a production machine, should continue after a power outage.

Retain variables, however, are reinitialized when executing the **Online** command **Reset origin** and, in contrast to persistent variables, when executing the **Online** command **Reset cold** or in the course of an application download.

NOTE: Only the specific variables defined as `VAR RETAIN` are stored in nonvolatile memory. However, local variables defined as `VAR RETAIN` in functions are NOT stored in nonvolatile memory. Defining `VAR RETAIN` locally in functions is of no effect.

Using interfaces or function blocks out of System Configuration libraries in the retain program section (`VAR_RETAIN`) will cause system exceptions, which may make the controller inoperable, requiring a re-start.

⚠WARNING

UNINTENDED EQUIPMENT OPERATION

- Do not use interfaces out of the SystemConfigurationItf library in the retain program section (`VAR_RETAIN`).
- Do not use function blocks out of the SystemConfiguration library in the retain program section (`VAR_RETAIN`).

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

NOTE: The libraries SystemConfigurationItf and SystemConfiguration are only available for PacDrive controllers (PacDrive LMC Eco, PacDrive LMC Pro/Pro2).

NOTE: If one variable in a function block is marked with `RETAIN`, then the function block instance with its variables is stored in remanent memory. This function is not available for all controllers. Consult the *Programming Guide* specific to your controller for detailed information.

Persistent Variables

Persistent variables are identified by keyword `PERSISTENT` (`VAR_GLOBAL PERSISTENT`). They are only reinitialized when executing the **Online** command **Reset origin**. In contrast to retain variables, they maintain their values after a download.

NOTE: Do not use the `AT` declaration in combination with `VAR PERSISTENT`.

Application example:

A counter for operating hours, which should continue counting even after a power outage or a download. Refer to the synoptic table on the behavior of remanent variables, page 573.

You can only declare persistent variables in a special global variable list of object type persistent variables, which is assigned to an application. You can add only one such list to an application.

NOTE: A declaration with `VAR_GLOBAL PERSISTENT` has the same effect as a declaration with `VAR_GLOBAL PERSISTENT RETAIN` or `VAR_GLOBAL RETAIN PERSISTENT`.

NOTE: Instant paths are added for variables that are marked with the `PERSISTENT` keyword outside the persistent list editor. For further information, refer to the **Add all instance paths** command (see Menu Commands, Online Help).

Like retain variables, the persistent variables are stored in a separate memory area.

Example

```
VAR_GLOBAL PERSISTENT RETAIN
  iVarPers1 : DINT; (* 1. Persistent+Retain Variable App1 *)
  bVarPers : BOOL;  (* 2. Persistent+Retain Variable App1 *)
END_VAR
```

NOTE: Persistent variables can only be declared inside the **Persistent list** object. If they are declared elsewhere, they will behave like retain variables and they will be reported as a detected **Build** error in the **Messages** view. (Retain variables can be declared in the global variable lists or in POUs.)

At each reload of the application, the persistent variable list on the controller will be checked against that of the project. The list on the controller is identified by the application. In case of inconsistencies, you will be prompted to reinitialize all persistent variables (see Menu Commands, Online Help) of the application. Inconsistency can result from renaming or removing or other modifications of the existing declarations in the list.

NOTE: Carefully consider any modifications in the declaration part of the persistent variable list and the effect of the results regarding reinitialization.

You can add new declarations only at the end of the list. During a download, these are detected as new and will not demand a reinitialization of the complete list. If you modify the name or data type of a variable, this is handled as a new declaration and provokes a reinitialization of the variable at the next online change or download.

NOTE: If one variable in a function block is marked with `PERSISTENT`, then the function block instance with its variables is stored in remanent memory, but only the one variable is treated as persistent.

Behavior of Remanent Variables

Consult the *Programming Guide* specific to your controller for further information on the behavior of remanent variables.

Constants - CONSTANT

Constants are identified by the keyword CONSTANT. You can declare them locally (programming object unit) or globally (global variable list). Constant variables can be accessed read-only in the implementation part of a POU.

Syntax

```
<scope> CONSTANT <identifier>:<type> := <initialization>;
END_VAR
```

With <scope> : VAR | VAR_INPUT | VAR_STAT | VAR_GLOBAL

Make sure to assign an initialization value when declaring a constant variable.

Example

```
VAR CONSTANT
c_iCon1:INT:=12; (* 1. Constant*)
END_VAR
```

Refer to the *Operands* chapter, page 746 for a list of possible constants.

Constants - VAR_GENERIC CONSTANT

A generic constant is a variable in the VAR_GENERIC CONSTANT declaration area of a function block which is not assigned an initialization value until the function block instance is assigned. The generic constant that is declared in the FUNCTION BLOCK can be used anywhere within the FUNCTION BLOCK as well as in its METHODS, PROPERTIES and ACTIONS, for example, in locally declared variables of parameters.

Syntax: Declaration of the function block:

```
FUNCTION_BLOCK <function block name>
VAR_GENERIC CONSTANT
    <generic constant name> : <integer data type> :=
<initial value>; //Initial value will be overwritten
END_VAR
;
```

You can use the generic constant (integer data type) within a function block for the size of arrays or the length of strings, for example. The initial value <initial value> is only needed for compile checks. At runtime, the value is overwritten.

> **NOTE:** You can also add a range for the integer data type. Refer to Subrange Types, page 666.

Syntax: Declaration of the function block instance:

```
PROGRAM PLC_PRG
VAR
    <fb instance name> : <function block name> < <literal>
>;
    <fb instance name> : <function block name> <(
<expression> )>;
    <array instance name> : ARRAY [ <index start > .. <index
end> ] OF <function block name> < <array length> > ;
END_VAR
;
```

When the function block instances are declared, the constant is assigned a specific value that is valid exclusively for this instance. For this purpose, the value <literal> in angle brackets is appended to the function block <function block name> that acts as the data type.

As an alternative, an expression <expression> can be appended. Enclose it in parentheses to enable usage of symbols such as < or > in an expression and help you preserve the uniqueness of the code.

Optionally, an ARRAY can be used to type the variable with an array of function blocks. The length of the array is <end index> – <start index> + 1. This is a function block with generic constants. The constants are used as array indices.

Example: arrMyString

< <array length> > assigns a specific array length to the data type that is valid specifically for this function block instance. The specified array length is inserted in angle brackets as a suffix to the data type.

Example: arrMyString : ARRAY[0..5] OF FB_MyString<6>;

Example: Using multiple generic constants:

```
FUNCTION_BLOCK FB_Pou
VAR_GENERIC CONSTANT
    lnstring : DINT := 10;
    numstring : DINT := 100;
END_VAR
VAR
    arraystring : ARRAY[0..numstring-1] OF STRING(lnstring);
END_VAR
;
PROGRAM PLC_PRG
VAR
    fbPou : FB_Pou<100, 1000>;
END_VAR
```

Example: Generic constant of a subrange type:

```
FUNCTION_BLOCK FB_SrString
VAR_GENERIC CONSTANT
    maxlen2 : INT(5..10) := 5;    //subrange data type
END_VAR
VAR
    arrTest : ARRAY[0..maxlen-1] OF BYTE;
END_VAR
;
PROGRAM SrMain
VAR CONSTANT
    cconst: INT(5..10) := 5;
END_VAR
VAR
    fbMyString1 : FB_SrString<5>;
    fbMyString2 : FB_SrString<(2 * cconst)>;
    arrMyString : ARRAY [0..5] OF FB_SrString<6>;
END_VAR
```

Example: Generic function block with parameterizable array variable:

The following code example demonstrates how to define a function block that can process arrays of arbitrary length. The function block has an array with a generic but constant length. In this context, the term constant means that although each function block instance varies in its array length, it remains constant during the lifetime of the object.

This can be useful in library programming, for example, for implementing a generic library POU.

```
FUNCTION_BLOCK FB_MyString
VAR_GENERIC CONSTANT
    maxlen : UDINT := 1;
END_VAR
VAR
    test : ARRAY[0..maxlen-1] OF BYTE;
END_VAR
;
PROGRAM PLC_PRG
VAR CONSTANT
    cconst: DINT := 1000;
END_VAR
VAR
    fbMyString1 : FB_MyString<100>;
    fbMyString2 : FB_MyString<(2 * cconst)>;
    arrMyString : ARRAY [0..5] OF FB_MyString<100>;
END_VAR
;
```

Inheritance

A function block with generic constants can also use the inheritance constructs `EXTENDS` and `IMPLEMENTS`.

> **NOTE:** Ensure you insert first the declaration of the generic constants followed by `EXTENDS` and `IMPLEMENTS` because generic constants can also be used with base classes.

Inheritance example:

```
INTERFACE IString
METHOD Append : BOOL
VAR_INPUT
    strAppend : IString;
END_VAR
METHOD Assign : BOOL
VAR_INPUT
    stringIn: String;
END_VAR
METHOD ToString : STRING
VAR_INPUT
END_VAR
```

```
PROPERTY Length : DINT
FUNCTION_BLOCK MyString
VAR_GENERIC CONSTANT
    maxlen : UDINT;
END_VAR
IMPLEMENTS IString
FUNCTION_BLOCK LongString EXTENDS MyString<1000>
FUNCTION_BLOCK MySpecialString
VAR_GENERIC CONSTANT
    maxlen2 : UDINT:= 1;
END_VAR
EXTENDS MyString<maxlen2>
METHOD ToLatin1: STRING
PROGRAM PLC_PRG
VAR CONSTANT
    cconst: DINT := 1000;
END_VAR
VAR
    string1 : MyString<100>;
    string2 : MyString<(2 * cconst)>;
    derived1 : LongString;
    derived2 : MySpecialString<100>;
END_VAR
string1.Assign ('Welt');
string2.Assign ('Hallo ');
string2.Append(string1);
derived2.ToLatin1('Hello World');
```

Typed Literals

Basically, in using IEC constants, the smallest possible data type will be used. If another data type has to be used, this can be achieved with the help of typed literals without the necessity of explicitly declaring the constants. For this, the constant will be provided with a prefix which determines the type.

Syntax

<type>#<literal>;

<type>	specifies the desired data type possible entries: BOOL, SINT, USINT, BYTE, INT, UINT, WORD, DINT, UDINT, DWORD, REAL, LREAL Write the type in uppercase letters.
<literal>	specifies the constant Enter data that fits within the data type specified under <type>.

Example

```
iVar1:=DINT#34;
```

If the constant cannot be converted to the target type without data loss, a message is issued.

You can use typed literals wherever normal constants can be used.

Constants in Online Mode

As long as the default setting **Replace constants** (**File > Project Settings > Compile options**) is activated, constants in online mode have a  symbol preceding the value in the **Value** column in the declaration or watch view. In this case, they cannot be accessed by, for example, forcing or writing.

VAR_CONFIG - Configuration Variables

Overview

You can use the variable configuration to map function block variables on the process image that is on the device I/Os. This avoids the need of specifying the definite address already in the declaration of the function block variable. The assignment of the definite address, page 760 in this case is done centrally for all function block instances in a global VAR_CONFIG list.

For this purpose, you can assign incomplete addresses to the function block variables declared between the keywords VAR and END_VAR. Use an asterisk to identify these addresses.

Identifier Syntax

<identifier> AT %<I|Q>* : <data type>

Example of the use of incompletely defined addresses:

```
FUNCTION_BLOCK locio
VAR
 xLocIn AT %I*: BOOL := TRUE;
 xLocOut AT %Q*: BOOL;
END_VAR
```

In this example, 2 local I/O variables are defined: a local input (%I*) and a local output variable (%Q*).

Define the addresses in the variable configuration in a global variable list (GVL) as follows:

Step	Action
1	Execute the **Add Object** command.
2	Add a **Global Variable List** (GVL) object to the **Devices Tree**.
3	Enter the declarations of the instance variables with the definite addresses between the keywords VAR_CONFIG and END_VAR.

When defining the addresses, note the following:

- Specify the instance variables by the complete instance path and separate the individual POUs and instance names from one another by periods.
- In the declaration, enter an address whose class (input/output) corresponds to that of the incomplete specified address (%I*, %Q*) in the function block.
- Verify that the data type agrees with the declaration in the function block.

Instance Variable Path Syntax

<instance variable path> AT %<I|Q><location> : <data type>;

Configuration variables whose instance path is invalid because the instance does not exist are denoted as detected errors. An error is also detected if no definite address configuration exists for an instance variable assigned to an incomplete address.

Example for a variable configuration

Assume that the following definition for function block locio - see the previous example - is given in a program:

```
PROGRAM PLC_PRG
VAR
locioVar1: locio;
locioVar2: locio;
END_VAR
```

Then a corrected variable configuration (in a global variable list) will be:

```
VAR_CONFIG
PLC_PRG.locioVar1.xLocIn AT %IX1.0 : BOOL;
PLC_PRG.locioVar1.xLocOut AT %QX0.0 : BOOL;
PLC_PRG.locioVar2.xLocIn AT %IX1.0 : BOOL;
PLC_PRG.locioVar2.xLocOut AT %QX0.3 : BOOL;
END_VAR
```

NOTE: Changes on directly mapped I/Os are immediately shown in the process image, whereas changes on variables mapped via `VAR_CONFIG` are not shown before the end of the responsible task.

NOTE: In a global variable list, the keywords `VAR_GLOBAL` and `VAR_CONFIG` can only be used exclusively.

Method Types

FB_Init, *FB_Reinit*, and *FB_Exit* Methods

General Purpose of the Methods

You can explicitly use the methods *FB_Init* and *FB_Reinit* to influence the initialization of function block variables as well as the behavior when exiting function blocks.

This chapter describes the methods, and the applications and effects of the methods in different conditions that require variable initialization.

FB_Init

By default, the `FB_Init` method is available implicitly. It is used to initialize a function block or a structure.

In order to influence the initialization, you can explicitly declare the `FB_Init` method by extending the given default initialization code. This allows you to evaluate the return value.

FB_Reinit

The *FB_Reinit* method must be declared explicitly.

If the *FB_Reinit* method is available, it is called after the instance of the corresponding function block has been copied (during an online change, page 580 after modifications in the function block declaration). It reinitializes the new instance module. The return value is not evaluated. In order to achieve that the base function block is reinitialized, call *FB_Reinit* explicitly for that function block. This allows you to evaluate the return value.

FB_Exit

The *FB_Exit* method must be declared explicitly.

If there is an implementation, then the method is called before the controller removes the code of the function block instance (implicit call). The return value is not evaluated.

The following paragraphs provide use cases of these methods for different operating conditions.

First Download

When you download an application to a controller that is in default state, the memory locations of the variables are set to the desired initial state. The data areas of function blocks are set to the desired values. You can influence this process by explicitly implementing *FB_Init* for function blocks in the program code of the application.

The method parameters *bInCopyCode* set to FALSE and *bInitRetains* set to TRUE indicate that a first download is being executed.

Online Change

When an **Online Change** command (see Menu Commands, Online Help) is executed, the methods *FB_Exit*, *FB_Init*, and *FB_Reinit* can be used to influence the initialization of function blocks.

During online change, the modifications made to the application in offline mode are downloaded to the controller. The instances of function blocks are updated by the new instances as follows:

If you have only made changes in the implementation part of the function block and not in the declaration part, the data areas are not replaced. The methods *FB_Init*, *FB_Reinit*, and *FB_Exit* are not called.

If you have made changes in the declaration part of a function block, the copy process described in the *FB_Reinit* paragraph, page 579 is performed when the **Online Change** command (see Menu Commands, Online Help) is executed. A list of the objects that have been changed since the last download is provided in the **Application Information** dialog box, page 244. Open this dialog box by clicking the **Details...** button in the dialog box where you select the option **Login with online change**.

The *FB_Init* and *FB_Reinit* method parameters *bInCopyCode* set to TRUE and *bInitRetains* set to FALSE indicate that an online change is being executed.

Calls Executed During an Online Change

Executing the **Online Change** command can change the contents of addresses.

⚠CAUTION

INVALID POINTER

Verify the validity of the pointers when using pointers on addresses and executing the Online Change command.

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

During an online change, the following calls are executed consecutively:

Step	Action	Comment
1	*FB_Exit*	`old_inst.FB_Exit(bInCopyCode := TRUE);` *FB_Exit* is called to initiate a cleanup process before the copy process is started. It prepares the data for the next copy process and influences the state of the new instance. Other parts of the application are informed about the position changes that are performed in the memory. Keep in mind that variables of type POINTER or REFERENCE keep their values during an online change and may no longer refer to the desired memory locations after the process has been completed. Variables of type INTERFACE are adapted during online change. External resources, such as socket, file, or other handles possibly can be adopted by the new instance, and often need no separate treatment during online change when the system resource is not affected by the online change copy process. If needed, this must be treated in *Init* or *Reinit* implementations.
2	*FB_Init*	`new_inst.FB_Init(bInitRetains := FALSE, bInCopyCode := TRUE);` *FB_Init* can be used to perform specific operations during the online change process. These are, for example, appropriately initializing variables at the new memory locations, or providing information on the new position of certain variables to other parts of the application.
3	Copy operation *copy*	`copy(&old_inst, &new_inst);` Existing values remain unchanged. For this purpose, they are copied from the old instance into the new instance.
4	*FB_Reinit*	`new_inst.FB_Reinit();` The *FB_Reinit* method is called after the copy operation. It sets the variables of the function block instance to defined values. For example, you can appropriately initialize variables at the new memory locations, or provide information on the new position of certain variables to other parts of the application. Implement the *FB_Reinit* method independent of online change because the method can be called by the application anytime, whenever a function block is to be reset to the original state.

NOTE: If you add the pragma, page 622 *{attribute no_copy}* to a variable of a function block, this variable will not be copied during online change; it will only be initialized.

Downloading an Updated Application

When you download an application to a controller that is already running an application, the existing application will be replaced. You can use the *FB_Exit* method, for example, to assign a defined state to external resources (such as socket or file handles).

The method parameters *bInCopyCode* set to FALSE and *bInitRetains* set to FALSE indicate that a download of an updated application is being executed.

Starting an Application

Before the first cycle of the tasks of an application is executed, the initial assignments are processed.

Example:

```
T1 : TON := (PT:=t#500ms);
```

The assignments are executed after *FB_Init* has been called. To be able to verify the impacts of these assignments, attach the *{attribute call_after_init}* pragma, page 599 to a function block and a method of a function block (for example, called *MyInit*). Insert this attribute above the declaration part of the function block and above the declaration part of the corresponding method. Attach this pragma also to function blocks that extend other function blocks which are using the *{attribute call_after_init}* pragma. It is a good practice to assign the same name, the same signature, and the same attribute to the corresponding method. To achieve this, call `SUPER^.MyInit`. Select a method name of your choice (except *FB_Init*, *FB_Reinit*, and *FB_Exit*). The method is called after the initial assignments have been processed and before the tasks of an application are started.

> **NOTE:** When the explicitly defined initialization code is executed, then the function block has already been initialized completely via the implicit initialization code. Due to this, calling `SUPER^.FB_Init` is not allowed.

Interface of the `FB_Init` Method

```
METHOD FB_Init : BOOL
VAR_INPUT
  bInitRetains : BOOL; // TRUE: the retain variables are
initialized (reset warm /reset cold)
  bInCopyCode : BOOL; // TRUE the instance will be copied to
the copy-code afterward (online change)
END_VAR
```

The return value is not used.

In an *FB_Init* method, you can declare additional function block inputs. Assign the inputs in the declaration of a function block instance.

Example: Method *FB_Init* for a function block `serialdevice`:

```
METHOD PUBLIC FB_Init : BOOL
VAR_INPUT
    bInitRetains : BOOL; // Initialization of the retain
variables
    bInCopyCode : BOOL; // Instance is copied to copy-code
    iCOMnum : INT; // additional input: number of the COM
interface that is to be observed
END_VAR
```

Instantiation of function block `serialdevice`:

```
    com1: serialdevice (iCOMnum:=1);
    com0: serialdevice (iCOMnum:=0);
```

Interface of the `FB_Reinit` Method

```
METHOD FB_Reinit : BOOL
```

Interface of the `FB_Exit` Method

The parameter *bInCopyCode.* is mandatory.

```
METHOD FB_Exit : BOOL
VAR_INPUT
bInCopyCode : BOOL; // TRUE: the exit method is called in
order to leave the instance which will be copied afterwards
(online change).
END_VAR
```

Derived Function Blocks

If a function block is derived from another function block, then the `FB_Init` method of the derived function block must define the same parameters as the `FB_Init` method of the base function block. However, you can add further parameters in order to implement a special initialization for the instance.

Example for the Call Order of Derived Function Blocks for `FB_Exit` and `FB_Init`

The following is assumed for the POUs named in this list: `SubFB EXTENDS MainFB` and `SubSubFB EXTENDS SubFB`:

Step	Action
1	`fbSubSubFb.FB_Exit(...);`
2	`fbSubFb.FB_Exit(...);`
3	`fbMainFb.FB_Exit(...);`
4	`fbMainFb.FB_Init(...);`
5	`fbSubFb.FB_Init(...);`
6	`fbSubSubFb.FB_Init(...);`

Pragma Instructions

Pragma Instructions

Overview

A pragma instruction is used to affect the properties of one or several variables concerning the compilation or precompilation (preprocessor) process. Therefore, a pragma influences the code generation.

> **NOTE:** Consider that the available pragmas are not 1:1 implementations of C preprocessor directives. They are handled as normal statements and therefore can only be used at statement positions. They must not be used within an expression and not in the declaration part of editors.

A pragma can determine whether a variable will be initialized, monitored, added to a parameter list, added to the symbol list, page 631, or made invisible in the **Library Manager**. It can force message outputs during the build process. You can use conditional pragmas to define how the variable should be treated depending on certain conditions. You can also enter these pragmas as definitions in the compile properties of a particular object.

You can use a pragma in a separate line, or with supplementary text in an implementation or declaration editor line. Within the FBD/LD/IL editor, execute the command **Insert Label** and replace the default text `Label:` in the arising text field by the pragma. In case you want to set a label as well as a pragma, insert the pragma first and the label afterwards.

The pragma instruction is enclosed in curly brackets.

Syntax

{ <instruction text> }

The opening bracket can immediately come after a variable name. Opening and closing brackets have to be in the same line.

Correct Positions for a Conditional Pragma

```
{IF defined(abc)}
IF x =abc THEN
  {IF defined(cde)}
    y := 12;
  {ELSE}
    y :=13;
  {END_IF}
END_IF
{ELSE}
IF x = 12 THEN
  {IF defined(cde)}
    y := 12;
  {ELSE}
    y :=13;
  {END_IF}
END_IF
```

Incorrect Positions for a Conditional Pragma

NOTE: Do **not** use conditional pragmas at positions indicated in this “bad practice” example.

```
{IF defined(abc)}
IF x = abc THEN
{ELSE}
IF x = 12 THEN
{END_IF}
y := {IF defined(cde)} 12; {ELSE} 13; {END_IF}
END_IF
```

Further Information

Depending on the type and contents of a pragma, the pragma operates on the subsequent statement, respectively all subsequent statements, until one of the following conditions is met:

- It is ended by an appropriate pragma.
- The same pragma is executed with different parameters.
- The end of the code is reached.

The term code in this context refers to a declaration part, implementation part, global variable list, or type declaration.

NOTE: Pragma instructions are case-sensitive.

If the compiler cannot meaningfully interpret the instruction text, the entire pragma is handled as a comment and is skipped.

Refer to the following pragma types:

- *Message Pragmas*, page 585
- *Attribute Obsolete*, page 627
- *Attribute Pragmas*, page 597
- *Conditional Pragmas*, page 587
- *Region Pragmas*, page 597
- *Attribute Symbol*, page 631

Message Pragmas

Overview

You can use message pragmas to force the output of messages in the **Messages** view (by default in the **Edit** menu) during the compilation (build) of the project.

You can insert the pragma instruction in an existing line or in a separate line in the text editor of a POU. Message pragmas positioned within currently not defined sections of the implementation code will not be considered when the project is compiled. For further information, refer to the example provided with the description of the defined (identifier) in the chapter *Conditional Pragmas*, page 587.

Types of Message Pragmas

There are 4 types of message pragmas:

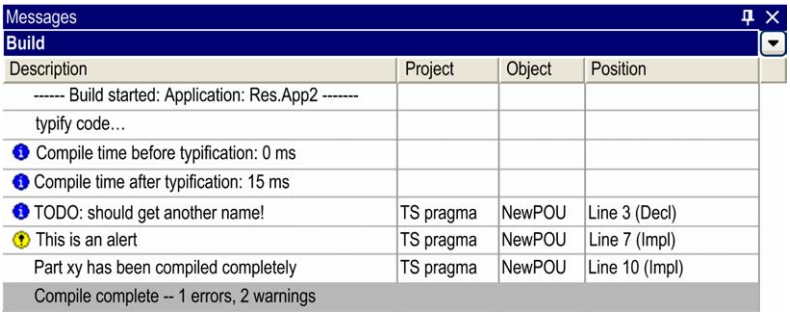
Pragma	Icon	Message Type
`{text 'text string'}`	–	text type The specified text string will be displayed.
`{info 'text string'}`		information The specified text string will be displayed.
`{warning digit 'text string'}`		alert type The specified text string will be displayed. In contrast to the global obsolete pragma, page 627, this alert is explicitly defined for the local position. **NOTE:** The alert pragma `{warning digit 'text string'}` is exclusive to objects such as POUs, to statements and to variables.
`{error 'text string'}`		error type The specified text string will be displayed.

NOTE: For messages of types information, alert, and detected error, you can reach the source position of the message - that is where the pragma is placed in a POU - by executing the command **Next Message**. This is not possible for the text type.

Example of Declaration and Implementation in ST Editor

```
VAR
ivar : INT; {info 'TODO: should get another name'}
bvar : BOOL;
arrTest : ARRAY [0..10] OF INT;
i:INT;
END_VAR
arrTest[i] := arrTest[i]+1;
ivar:=ivar+1;
{warning 'This is an alert'}
{text 'Part xy has been compiled completely'}
```

Output in **Messages** view:



Conditional Pragmas

Overview

The ExST (Extended ST) language supports several conditional *Pragma Instructions*, page 583, which affect the code generation in the precompile or compile process.

NOTE: Do not use any conditional pragmas in the declaration part. They are not regarded. The conditional pragma with the `{project_define (<global define>)}`, page 593 operator is an exception. You can use it in the implementation part and declaration part of POUs.

The implementation code which will be regarded for compilation can depend on the following conditions:

- Is a certain data type or variable declared?
- Does a type or variable have a certain attribute?
- Does a variable have a certain data type?
- Is a certain POU or task available or is it part of the call tree, etc...

NOTE: It is not possible for a POU or GVL declared in the **POUs Tree** to use a `{define...}` declared in an application. Definitions in applications will only affect interfaces inserted below the respective application.

`{define identifier string}`	During preprocessing, all subsequent instances of the identifier will be replaced with the given sequence of tokens if the token string is not empty (which is allowed and well-defined). The identifier remains defined and in scope until the end of the object or until it is undefined in an `{undefine}` directive. Used for conditional compilation, page 588.
`{undefine identifier}`	The preprocessor definition of the `identifier` (by `{define}`, see first row of this table) will be removed and the identifier hence is undefined. If the specified identifier is not currently defined, this pragma will be ignored.

`{IF expr}`
...
`{ELSIF expr}`
...
`{ELSE}`
...
`{END_IF}`
	These are pragmas for conditional compilation. The specified expressions `exprs` are required to be constant at compile time; they are evaluated in the order in which they appear until one of the expressions evaluates to a non-zero value. The text associated with the successful directive is preprocessed and compiled normally; the others are ignored. The order of the sections is determinate; however, the `elsif` and `else` sections are optional, and `elsif` sections may appear arbitrarily more often.
	Within the constant `expr`, you can use several conditional compilation operators, page 588.

`<expr>`	Within the constant expression `expr` of a conditional compilation pragma (`{if}` or `{elsif}`) (see previous table), you can use several operators. These operators may not be undefined or redefined via `{undefine}` or `{define}`, respectively.

You can also use these expressions as well as the definition completed by `{define}` in the **Compiler defines:** text field in the **Properties** dialog box of an object (**View > Properties > Build**).

Conditional Compilation Operators

The following operators are supported:

- defined (identifier), page 588
- `defined (variable:<variable name>)`, page 589
- defined (type:identifier), page 589
- defined (pou:pou-name), page 589
- `defined (task:task-name)`, page 590
- hasattribute (pou: pou-name, attribute), page 590
- hasattribute (variable: variable, attribute), page 591
- hastype (variable:variable, type-spec), page 592
- hasvalue (define-ident, char-string), page 593
- `project_defined (<global define>)`, page 593
- `hasvalue (PackMode, '<pack mode value>')`, page 593
- `hasvalue (PackMode, '<pack mode value>')`, page 593
- `hasconstanttype(<constant name>, <boolean literal>)`, page 594
- hasconstantvalue (variable, literal expression), page 595
- NOT operator , page 596
- operator AND operator , page 596
- operator OR operator , page 596
- operator , page 596

defined (identifier)

This operator affects that the expression gets value TRUE, as soon as the `identifier` has been defined with a `{define}` instruction and has not been undefined later by an `{undefine}` instruction. Otherwise its value is FALSE.

Example on `defined (identifier)`:

Precondition: There are 2 applications `App1` and `App2`. Identifier `pdef1` is defined in `App2`, but not in `App1`.

```
{IF defined (pdef1)}
    (* this code is processed in App1 *)
    {info 'pdef1 defined'}
    hugo := hugo + SINT#1;
{ELSE}
    (* the following code is only processed in application
App2 *)
    {info 'pdef1 not defined'}
    hugo := hugo - SINT#1;
{END_IF}
```

Additionally, an example for a message pragma, page 585 is included:

Only information `pdef1 defined` will be displayed in the **Messages** view when the application is compiled because `pdef1` is defined. The message **pdef1 not defined** will be displayed when `pdef1` is not defined.

defined (variable:<variable name>)

When applied to a variable, its value is TRUE if this particular variable is declared within the current scope. Otherwise it is FALSE.

Example on `defined (variable:<variable name>)`:

Precondition: There are 2 applications `App1` and `App2`. Variable `g_bTest` is declared in `App2`, but not in `App1`.

```
{IF defined (variable:g_bTest)}
(* the following code is only processed in application App2
*)
g_bTest := x > 300;
{END_IF}
```

defined (type:identifier)

When applied to a type identifier, its value is TRUE if a type with that particular name is declared. Otherwise it is FALSE.

Example on `defined (type:identifier)`:

Precondition: There are 2 applications `App1` and `App2`. Data type DUT is defined in `App2`, but not in `App1`.

```
{IF defined (type:DUT)}
(* the following code is only processed in application App1
*)
bDutDefined := TRUE;
{END_IF}
```

defined (pou:pou-name)

When applied to a POU name, its value is TRUE if one of the following objects with the name `<pou-name>` is available:

- Function block
- Function
- Program
- Action
- Method
- Interface

Otherwise it is FALSE.

For more information, refer to the chapter POU, page 173.

Example on `defined (pou: pou-name)`:

Precondition: There are 2 applications `App1` and `App2`. POU `CheckBounds` is available in `App2`, but not in `App1`.

```
{IF defined (pou:CheckBounds)}
(* the following code is only processed in application App1
*)
arrTest[CheckBounds(0,i,10)] := arrTest[CheckBounds(0,i,10)]
+ 1;
{ELSE}
(* the following code is only processed in application App2
*)
arrTest[i] := arrTest[i]+1;
{END_IF}
```

defined (task:task-name)

The expression gets the value TRUE if a task is defined with the name <task name>. Otherwise it is FALSE.

Syntax:

```
{ IF defined (task: <task name> }
{ ELSIF defined (task: <task name> }
```

Example 1: {IF defined (task: Task_D)}

Example 2:

Precondition: There are 2 applications App1 and App2. The task PLC_PRG_Task is defined in App1, but not in App2.

```
{IF defined (task: PLC_PRG_Task)}
    (* the following code is only processed in App1 *)
    erg := plc_prg.x;
{ELSE}
    (* the following code is only processed in App2 *)
    erg := prog.x;
{END_IF}
```

hasattribute (pou: pou-name, attribute)

When applied to a POU, its value is TRUE if this particular attribute is specified in the first line of the POUs declaration part.

Example on hasattribute (pou: pou-name, attribute):

Precondition: There are 2 applications App1 and App2. Function *fun1* is defined in App1 and App2, but in App1 has an attribute *vision*:

Definition of fun1 in App1:

```
{attribute 'vision'}
FUNCTION fun1 : INT
VAR_INPUT
i : INT;
END_VAR
VAR
END_VAR
```

Definition of fun1 in App2:

```
FUNCTION fun1 : INT
VAR_INPUT
i : INT;
END_VAR
VAR
END_VAR
```

Pragma instruction

```
{IF hasattribute (pou: fun1, 'vision')}
(* the following code is only processed in application App1
*)
ergvar := fun1 ivar);
{END_IF}
```

hasattribute (variable: variable, attribute)

When applied to a `variable`, its value is TRUE if this particular attribute is specified via the *{attribute}* instruction in a line before the declaration of the variable.

Example on `hasattribute (variable: variable, attribute)`:

Precondition: There are 2 applications `App1` and `App2`. Variable *g_globalInt* is used in `App1` and `App2`, but in `App1` has an attribute *DoCount* :

Declaration of *g_globalInt* in `App1`

```
VAR_GLOBAL
{attribute 'DoCount'}
g_globalInt : INT;
g_multiType : STRING;
END_VAR
```

Declaration of *g_globalInt* in `App2`

```
VAR_GLOBAL
g_globalInt : INT;
g_multiType : STRING;
END_VAR
```

Pragma instruction

```
{IF hasattribute (variable: g_globalInt, 'DoCount')}
(* the following code line will only be processed in App1,
because there variable g_globalInt has got the attribute
'DoCount' *)
g_globalInt := g_globalInt + 1;
{END_IF}
```

hastype (variable:variable, type-spec)

When applied to a `variable`, its value is TRUE if this particular variable has the specified `type-spec`. Otherwise it is FALSE.

Available data types of `type-spec`

- BOOL
- BYTE
- DATE
- DATE_AND_TIME
- DINT
- DWORD
- INT
- LDATE
- LDATE_AND_TIME (LDT)
- LINT
- LREAL
- LTIME
- LTIME_OF_DAY (LTOD)
- LWORD
- REAL
- SINT
- STRING
- TIME
- TIME_OF_DAY (TOD)
- UDINT
- UINT
- ULINT
- USINT
- WORD
- WSTRING

Example on operator *hastype (variable: variable, type-spec)*:

Precondition: There are 2 applications `App1` and `App2`. Variable *g_multitype* is declared in `App1` with type LREAL and in application `App2` with type STRING:

```
{IF (hastype (variable: g_multitype, LREAL))}
(* the following code line will be processed only in App1 *)
g_multitype := (0.9 + g_multitype) * 1.1;
{ELSIF (hastype (variable: g_multitype, STRING))}
(* the following code line will be processed only in App2 *)
g_multitype := 'this is a multitalent';
{END_IF}
```

hasvalue (define-ident, char-string)

If the define (*define-ident*) is defined and it has the specified value (*char-string*), then its value is TRUE. Otherwise it is FALSE.

Example on `hasvalue (define-ident, char-string)`:

Precondition: Variable *test* is used in applications `App1` and `App2`. It gets value 1 in `App1` and value 2 in `App2`:

```
{IF hasvalue(test,'1')}
(* the following code line will be processed in App1,
because there variable test has value 1 *)
x := x + 1;
{ELSIF hasvalue(test,'2')}
(* the following code line will be processed in App1,
because there variable test has value 2 *)
x := x + 2;
{END_IF}
```

project_defined (<global define>)

The expression gets the value TRUE if the global define `<global define>` is included in the list of project defines in the **Project Settings - Compile Options** dialog box (for more information, refer to the Menu Commands Online Help).

The operator is specified as a condition of an `IF` statement and can be used both in the implementation part and in the declaration part of ST POUs and libraries. It can be combined with other defines by means of the `AND` and `OR` operators if they are allowed at the respective position.

In the declaration part of POUs, `IF` statements with the operator `project_defined (<global define>)` can contain the following constructs:

- Variable declarations

  Example:

  ```
  {If project_defined(define1)}
      x : DINT;
  {END_IF}
  ```

- Comments
- Attribute declarations
- Pragma statements

The following constructs are not allowed:

- Full declarations of POUs are not allowed.

  Negative example:

  ```
  {If project_defined(define1)}
  FUNCTION POU : BOOL
  VAR_INPUT
      x : INT;
  END_VAR
  {END_IF}
  ```

- Scopes of variables are not allowed: `VAR_INPUT`, `VAR_OUTPUT`, `VAR_IN_OUT`

hasvalue (PackMode, '<pack mode value>')

The pack mode that is verified depends on the device description, not on the pragma that can be specified for individual DUTs.

hasvalue (RegisterSize, '<register size>')

<register size>: Defines the size of a CPU register (in bits).

When the size of a CPU register is equal to <register size>, the expression returns the value TRUE .

<register size> can have the following values:

- 16 for C16x
- 64 for X86-64 bit
- 32 for X86-32 bit

hasconstanttype(<constant name>, <boolean literal>)

The operator verifies whether the constant, which is defined with <constant name>, has been replaced. The second parameter (Boolean value) controls what is verified:

- TRUE: Verifies whether the constant has been replaced.
- FALSE: Verifies whether the constant has not been replaced.

If the respective case is detected, the operator returns TRUE.

Syntax:

```
{ IF hasconstanttype( <constant name> , <boolean literal> )
}
{ ELSIF hasconstanttype( <constant name> , <boolean literal>
) }
```

Example:

```
{IF hasconstanttype(PLC_PRG.aConst, TRUE)}
```

The automatic replacement of constants depends on the following:

- Compile option **Replace constants**
- Constant type (STRING types, for example, are not replaced)
- Usage of the attribute {attribute 'const_non_replaced'}
- Usage of the attribute {attribute 'const_replaced'}

Example:

```
VAR
    iCntMAXIsReplaced: INT;
    xErrorOccured : BOOL;
END_VAR
VAR CONSTANT
    c_iMAX: INT := 99;
END_VAR

{IF hasconstanttype(c_iMAX, TRUE)}
    iCntMAXIsReplaced := iCntMAXIsReplaced + 1;  // E.g.
this code will be executed in case Project Settings -
Compile options - Replace constants was activated
{ELSE}
    xErrorOccured := FALSE;;                    // E.g. this
code will be executed in case Project Settings - Compile
options - Replace constants was deactivated
{END_IF}
```

hasconstantvalue(<constant name>, <variable name>, <comparison operator>)

The operator compares the value of the constant, which is defined with <constant name>, with the value of the second parameter. The second parameter can be specified either as a literal <literal> or as variable <variable name>.

The following comparison operators <comparison operator> are supported:

- Greater than (>)
- Greater than or equal to (>=)
- Equal to (=)
- Not equal to (<>)
- Less than or equal to (<=)
- Less than (<)

Syntax:

```
{ IF hasconstantvalue( <constant name> , <variable name> ,
<comparison operator> )
{ IF hasconstantvalue( <constant name> , <literal> ,
<comparison operator> )
{ ELSIF hasconstantvalue( <constant name> , <variable name>
, <comparison operator> )
{ ELSIF hasconstantvalue( <constant name> , <literal> ,
<comparison operator> )
```

Example 1:

```
{IF hasconstantvalue(PLC_PRG.aConst, 99, >)}

{ELSIF hasconstantvalue(PLC_PRG.aConst, GVL.intconst99, =)}
```

Example 2:

```
PROGRAM PRG_ConditionConstantValue
VAR
    iCntMAX: INT;
    iCntGlobalMAX : INT;
    iCntABC: INT;
    iCntGlobalABC : INT;
    xErrorOccured : BOOL;
END_VAR
VAR CONSTANT
    c_iMAX: INT := 999;
    c_sABC: STRING := 'ABC';
    {attribute 'const_non_replaced'}
    c_iNonReplaceable: INT := 888;
END_VAR

{IF hasconstantvalue(c_iMAX, 999, =)}
    iCntMAX := iCntMAX + 1;
{ELSE}
    xErrorOccured := FALSE;
{END_IF}

{IF hasconstantvalue(c_iMAX, GVL.gc_iMAX, =)}
    iCntGlobalMAX := iCntGlobalMAX + 1;
{ELSE}
    xErrorOccured := FALSE;
{END_IF}

{IF hasconstantvalue(c_sABC, 'ABC', =)}
    iCntABC := iCntMAX + 1;
{ELSE}
    xErrorOccured := FALSE;
{END_IF}
{IF hasconstantvalue(c_sABC, GVL.gc_sABC, =)}
    iCntGlobalABC := iCntMAX + 1;
```

```
{ELSE}
    xErrorOccured := FALSE;
{END_IF}
```

NOT operator

The expression gets value TRUE when the inverted value of `operator` is TRUE. `operator` can be one of the operators described in this chapter.

Example on `NOT operator`:

Precondition: There are 2 applications `App1` and `App2`. POU *PLC_PRG1* is used in `App1` and *App2*. POU `CheckBounds` is only available in `App1`:

```
{IF defined (pou: PLC_PRG1) AND NOT (defined (pou:
CheckBounds))}
(* the following code line is only executed in App2 *)
bANDNotTest := TRUE;
{END_IF}
```

AND operator

The expression gets value TRUE if both operators are TRUE. `operator` can be one of the operators listed in this table.

Example on `AND operator`:

Precondition: There are 2 applications `App1` and `App2`. POU `PLC_PRG1` is used in applications `App1` and `App2`. POU `CheckBounds` is only available in `App1`:

```
{IF defined (pou: PLC_PRG1) AND (defined (pou:
CheckBounds))}
(* the following code line will be processed only in
applications App1, because only there "PLC_PRG1" and
"CheckBounds" are defined *)
bORTest := TRUE;
{END_IF}
```

OR operator

The expression is TRUE if one of the operators is TRUE. `operator` can be one of the operators described in this chapter.

Example on `OR operator`:

Precondition: POU `PLC_PRG1` is used in applications `App1` and `App2`. POU `CheckBounds` is only available in `App1`:

```
{IF defined (pou: PLC_PRG1) OR (defined (pou: CheckBounds))}
(* the following code line will be processed in applications
App1 and App2, because both contain at least one of the POUs
"PLC_PRG1" and "CheckBounds" *)
bORTest := TRUE;
{END_IF}
```

(operator)

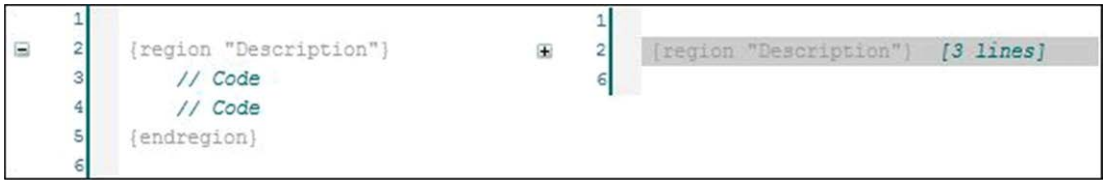
`(operator)` braces the operator.

Region Pragmas

Overview

Use region pragmas to group several lines into one block in a text editor. You can assign a name to the block. Region pragmas can be nested.

The figure shows a program code that contains a region pragma in the extended and in the collapsed view.



Region pragmas can be used in the ST editor and in the declaration editors.

Attribute Pragmas

Attribute Pragmas

Overview

You can assign attribute pragmas, page 583 to a signature in order to influence the compilation or pre-compilation that is the code generation.

There are user-defined attributes, page 597, which you can use in combination with conditional pragmas, page 587.

Attributes are defined within the declaration part. An exception is made for the action and transition objects which do not have a declaration part. You can define the attributes at the beginning of the implementation part.

User-Defined Attributes

Overview

You can assign arbitrary user-defined or application-defined attribute pragmas to POUs, type declarations, or variables. This attribute can be queried before compilation by conditional pragmas, page 587.

Syntax

{attribute 'attribute'}

This pragma instruction is valid for the subsequent POU declaration or variable declaration.

You can assign a user-defined attribute to:

- a POU or action
- a variable
- a data type

Example on POUs and Actions

Attribute `vision` for function `fun1`:

```
{attribute 'vision'}
FUNCTION fun1 : INT
VAR_INPUT
i : INT;
END_VAR
VAR
END_VAR
```

Example on Variables

Attribute `DoCount` for variable `ivar` :

```
PROGRAM PLC_PRG
VAR
{attribute 'DoCount'};
ivar:INT;
bvar:BOOL;
END_VAR
```

Example on Types

Attribute `aType` for data type DUT_1:

```
{attribute 'aType'}
TYPE DUT_1 :
STRUCT
a:INT;
b:BOOL;
END_STRUCT
END_TYPE
```

For the usage of conditional pragmas, refer to the chapter *Conditional Pragmas*, page 587.

Attribute call_after_global_init_slot

Overview

All functions and programs containing this attribute in an own line above their declaration part are called after the global initialization (`GlobalInit`). The calling sequence is determined by the attribute value.

> **NOTE:** Compile errors will be detected (during code generation) if *VAR_INPUT* declarations are used in functions or methods that contain this attribute. The reason is that the input variables are unknown when the function is called implicitly during online change.

Syntax

{attribute 'call_after_global_init_slot' := '<slot>'}

Replace `<slot>` by an integer value defining the priority within the calling sequence: The lower the value, the earlier the call. In case of several signatures carrying the same value for the attribute, the sequence of their initialization remains undefined.

If a method is provided with the attribute, then it is called for all instances of the concerned function block. All instances are called within the specified slot; however, you cannot control the order among the instances themselves.

Attribute call_after_init

Overview

Use the pragma `{attribute call_after_init}` to define a method that is called implicitly after the initialization of a function block instance. For performance reasons, attach the attribute both to the function block itself and to the instance method to be called. The method has to be called after FB_Init, page 582 and after having applied the variable values of an initialization expression in the instance declaration.

NOTE: Compile errors will be detected if *VAR_INPUT* declarations are used in methods that contain this attribute. The reason is that the input variables are unknown when the method is called implicitly during online change.

Syntax

{attribute 'call_after_init'}

Example

With the following definition:

```
{attribute 'call_after_init'}
FUNCTION_BLOCK FB
... <functionblock definition>
{attribute 'call_after_init'}
METHOD FB_AfterInit
... <method definition>
```

... declaration like the following:

```
inst : FB := (in1 := 99);
```

... will result in the following order of code processing:

```
inst.FB_Init();
inst.in1 := 99;
inst.FB_AfterInit();
```

So, in `FB_Afterinit`, you can react on the user-defined initialization.

Attribute call_after_online_change_slot

Overview

All functions and programs containing this attribute in an own line above their declaration part are called after an online change. The calling sequence is determined by the attribute value.

NOTE: Compile errors will be detected (during code generation) if *VAR_INPUT* declarations are used in functions or methods that contain this attribute. The reason is that the input variables are unknown when the function is called implicitly during online change.

Syntax

{attribute 'call_after_online_change_slot' := '<slot>'}

Replace `<slot>` by an integer value defining the priority within the calling sequence: The lower the value, the earlier the call. If several modules have the same priority value for the attribute, then the order in which they are called remains undefined.

If a method is provided with the attribute, then it is called for all instances of the concerned function block. All instances are called within the specified slot; however, you cannot control the order among the instances themselves.

NOTE: As the application cannot run during an online change, each code executed in this situation can effect jitter. For this reason, minimize the code to be executed.

Attribute call_before_global_exit_slot

Overview

All functions and programs containing this attribute in an own line above their declaration part are called after the `GlobalExit`. The `GlobalExit` is executed before a new download, at a reset, or during an online change and affects modules which are provided with an `FB_exit` method. The calling sequence is determined by the attribute value.

NOTE: Compile errors will be detected (during code generation) if *VAR_INPUT* declarations are used in functions or methods that contain this attribute. The reason is that the input variables are unknown when the function is called implicitly during online change.

Syntax

{attribute 'call_before_global_exit_slot' := '<slot>'}

Replace `<slot>` by an integer value defining the priority within the calling sequence: The lower the value, the earlier the call. If several modules have the same priority value for the attribute, then the order in which they are called remains undefined.

If a method is provided with the attribute, then it is called for all instances of the concerned function block. All instances are called within the specified slot; however, you cannot control the order among the instances themselves.

Attribute call_on_type_change

Overview

Attach the *Attribute call_on_type_change* pragma to methods of a function block *A* in order to achieve that this method is called when the data type is changed for one or more function blocks *B*, *C*, etc. that are referenced by *A*. The function blocks can be referenced by pointers, page 647 or references, page 646.

Syntax

{attribute 'call_on_type_change':= '<name of the first referenced function block>|<name of the second referenced function block>|<name of the nth referenced function block>'}

Insert the *Attribute call_on_type_change* above the first line in the method declaration.

Examples

Example of a function block with references:

```
FUNCTION_BLOCK FB_A
...
```

```
VAR
    var_pt: POINTER TO FB_B;
    var_ref: REFERENCE TO FB_C;
END_VAR
...
```

Example of a method that is called when data types are changed in the referenced function blocks *FB_B* and *FB_C*:

```
{attribute 'call_on_type_change' := 'FB_B,
FB_C'}
METHOD METH_react_on_type_change : INT
VAR_INPUT
...
```

Attribute conditionalshow

Overview

The function block or variable that contains this attribute is not displayed in the user interface if the library is installed as **.compiled-library*.

If the library is available as source library (**.library*) in the repository, the function block or variable is visible in the user interface.

If you use `conditionalshowsymbols` as a command-line option, such as `LogicBuilder.exe --conditionalshowsymbols="Library_Developer"`, the function blocks or variables are visible in the user interface. This option may be useful for library developers when testing libraries. For more information, refer to Command Line Arguments Activating Visibility, page 602.

Syntax

```
{attribute 'conditionalshow' := 'SomeText'}
```

or

```
{attribute 'conditionalshow'}
```

Insert Location

Insert this pragma in the first line of the declaration part of the function block or before a variable.

Example

Example for hiding a variable:

```
FUNCTION_BLOCK FB_DataManager
VAR
    {attribute 'conditionalshow' := 'Library_Developer'}
    iLocal : INT;
    iCounter : INT;
END_VAR
```

Result: The variable `iLocal` is invisible.

Example for hiding a function block:

```
{attribute 'conditionalshow' := 'Library_Developer'}
FUNCTION_BLOCK FB_DataManager
VAR
    iLocal : INT;
    iCounter : INT;
END_VAR
```

Result: The identifiers `FB_DataManager`, `iLocal`, and `iCounter` are invisible.

Command Line Arguments Activating Visibility

You can unhide hidden variables without the source library (**.library*) being available in the repository by using the command line argument `conditionalshowsymbols`. To enable visibility, enter the attribute values of the pragma separated by commas.

Syntax:

```
LogicBuilder.exe --conditionalshowsymbols=" <some text> ( ,
<next text> )* "
LogicBuilder.exe --conditionalshowsymbols="Library_
Developer"
LogicBuilder.exe --conditionalshowsymbols="Group_A,Group_B"
```

The following table illustrates the behavior when using the compiled library containing `FB_DataManager` of the above examples, whereby FB1 is used as abbreviation for the first and FB2 for the second `FB_DataManager` example. fb1 and fb2 are instances of FB1 and FB2. Additionally, it is assumed a library containing a function `Func1` that is tagged with `{attribute 'conditionalshow' := 'Library_Developer'}`.

	Starting Logic Builder without option	**Starting Logic Builder `--conditionalshowsymbols= "Library_Developer")`**
Input Assistant	• FB1 is displayed. • FB2 is not displayed. • `Func1` is not displayed. • Variables of fb1 and fb2 are not displayed.	• FB1 is displayed. • FB2 is displayed. • `Func1` is displayed. • Variables of fb1 and fb2 are not displayed.(1)
SmartCoding	• FB1 is displayed. • FB2 is not displayed. • `Func1` is not displayed. • Variables of fb1 and fb2 are not displayed.	• FB1 is displayed. • FB2 is displayed. • `Func1` is displayed. • Variables of fb1 and fb2 are not displayed.(1)
Library Manager	• FB1 is displayed. • FB2 is not displayed. • `Func1` is not displayed.	• FB1 is displayed. • FB2 is displayed. • `Func1` is displayed.
Monitoring	• Only `iCounter` of fb1 is displayed. • Both variables of fb2 are displayed.(2)	Both variables of fb1 and fb2 are displayed.
Symbol Configuration	• Only `iCounter` of fb1 is displayed. • No variables of fb2 are displayed.	Both variables of fb1 and fb2 are displayed (and can be selected, for example, for data exchange).

(1) Variables declared as VAR in the scope of a function block are for internal usage and therefore not displayed in the **Input Assistant** or **SmartCoding**.

(2) As FB2 is not displayed in the **Input Assistant**, **SmartCoding** or the **Library Manager**, it is not known how to declare fb2 as FB2. Therefore, monitoring of fb2 is not applicable.

Attribute conditionalshow_all_locals

Overview

The local variables are not displayed in the user interface if the integrated library is installed as **.compiled-library*.

If the integrated library is available as **.library* in the repository, the local variables of the function block are visible in the user interface.

If you use `conditionalshow_all_locals` as a command-line option, such as `LogicBuilder.exe --conditionalshow_all_locals= "libdeveloper"."--conditionalshow="libdeveloper"`, the local variables of the function blocks are visible in the user interface. This option may be useful for library developers when testing libraries.

Syntax

```
{attribute 'conditionalshow_all_locals' :='SomeText'}
```

or

```
{attribute 'conditionalshow_all_locals'}
```

Insert Location

Insert this pragma in the first line of the declaration part of the function block.

Example

```
{attribute 'conditionalshow_all_locals' := 'SomeText'}
//{attribute 'conditionalshow_all_locals'}
FUNCTION_BLOCK FB_Test
VAR_INPUT
END_VAR
VAR_OUTPUT
    eErrorID : ERROR;
VAR
    iLocal : INT;
END_VAR
```

Attribute const_replaced, Attribute const_non_replaced

Overview

Insert the pragma `{attribute 'const_replaced'}` in the declaration of a global constant of type scalar if you explicitly want to activate the compiler option **Replace constants** for this constant. This has the effect that the constant becomes available in the **Symbol Configuration**.

> **NOTE:** The pragma does not have an effect on variables of compound types, such as arrays and structures.

Correspondingly, you can insert the pragma `{attribute 'const_non_replaced'}` in order to deactivate the compiler option **Replace constants**.

The option **Replace constants** is pre-defined for the whole project in the **Project Settings > Compile options** dialog box (see Menu Commands, Online Help).

Syntax

```
{attribute 'const_replaced'}
{attribute 'const_non_replaced'}
```

Example

The constants `iTestCon` and `bTestCon` are available in the **Symbol Configuration** because **Replace constants** is deactivated by pragmas.

```
VAR_GLOBAL CONSTANT
    {attribute 'const_non_replaced'}
    iTestCon    :    INT  := 12;
    {attribute 'const_non_replaced'}
    bTestCon    :    BOOL := TRUE;
    rTestCon    :    REAL := 1.5;
END_VAR
VAR_GLOBAL
    iTestVar    :    INT  := 12;
    bTestVar    :    BOOL := TRUE;
END_VAR
```

Attribute 'dataflow'

Overview

The pragma *{attribute 'dataflow'}* enables you to control the dataflow when processing function blocks in the FBD/LD/IL editor. The attribute defines the input or output of a function block that is used as the connection to the next or previous function block.

You can assign this attribute to only one input and one output in the declaration of the function block.

For function blocks without the *{attribute 'dataflow'}*, the dataflow is determined automatically as follows:

The connection is established between an output and an input of the same type. The uppermost input and output variables of the function block are used first. If there are no variables with the same data type, then the uppermost output is connected to the uppermost input of the next function block.

You can also change the control flow in the editor by using the pointer to connect the connector pins of the function block to other positions. For further information, refer to the description of *Inserting, Arranging, and Replacing Elements*, page 275.

Syntax

```
{attribute 'dataflow'}
```

Example

The `FB` and the previous function block are connected using the input variable i1. The `FB` and the next function block are connected using the output variable `outRes1`.

```
FUNCTION_BLOCK FB
VAR_INPUT
            r1 : REAL;
            {attribute 'dataflow'}
            i1 : INT;
            i2 : INT;
```

```
            r2 : REAL;
END_VAR
VAR_OUTPUT
            {attribute 'dataflow'}
            outRes1 : REAL;
            out1 : INT;
            g1 : INT;
            g2 : REAL;
END_VAR
```

Attribute displaymode

Overview

Use the pragma *{attribute displaymode}* to define the display mode of a single variable. This setting will overwrite the global setting for the display mode of all monitoring variables done via the commands of the submenu **Display Mode** (by default in the **Online** menu).

Position the pragma in the line above the line containing the variable declaration.

Syntax

{attribute 'displaymode':=<displaymode>}

The following definitions are possible:

- to display in binary format

```
{attribute 'displaymode':='bin'}
{attribute 'displaymode':='binary'}
```

- to display in decimal format

```
{attribute 'displaymode':='dec'}
{attribute 'displaymode':='decimal'}
```

- to display in hexadecimal format

```
{attribute 'displaymode':='hex'}
{attribute 'displaymode':='hexadecimal'}
```

Example

```
VAR
   {attribute 'displaymode':='hex'}
   dwVar1: DWORD;
END_VAR
```

Attribute enable_dynamic_creation

Overview

The pragma `enable_dynamic_creation` is required for using the `__NEW` operator, page 728 for function blocks.

Syntax

{attribute 'enable_dynamic_creation'}

Insert the pragma in the first line in the declaration of the function block.

Attribute estimated-stack-usage

Overview

The pragma *{attribute 'estimated-stack-usage' := '<estimated stack size in bytes>'}* helps to prevent runtime systems with an active stack check from issuing a message (`C0298: Calculation of stack usage incomplete because of recursive calls`) indicating that there is insufficient space in the stack. This check is performed during the code generation. For recursive methods, you may want to reduce the number of messages.

Syntax

```
{attribute 'estimated-stack-usage' := '<estimated stack size
in bytes>'}
```

Insert Location

Insert this pragma in the line above the `METHOD` declaration in the declaration section of the relevant method.

Example

```
{attribute 'estimated-stack-usage' := '99'}
METHOD PUBLIC m_Temp : UDINT
VAR_INPUT
    uiN : UINT;
END_VAR

m_Temp := 1;
IF uiN > 1 THEN
    m_Temp := uiN * THIS^.m_Temp(uiN := (uiN- 1));
    RETURN;
ELSE
    RETURN;
END_IF
```

Attribute ExpandFully

Overview

Use the pragma `{attribute 'ExpandFully'}` to make all members of arrays used as input variables for referenced visualizations accessible within the **Visualization Properties** dialog box.

Syntax

{attribute 'ExpandFully'}

Example

Visualization *visu* is intended to be inserted in a frame within visualization *visu_main*.

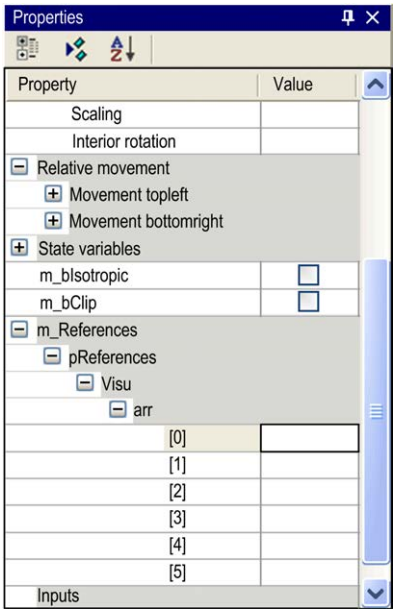
As input variable *arr* is defined in the interface editor of *visu* and will later be available for assignments in the **Properties** dialog box of the frame in `visu_main`.

In order to get the available particular components of the array in the **Properties** dialog box, insert the attribute *ExpandFully* in the interface editor of *visu* directly before *arr*.

Declaration in the interface editor of *visu*:

```
VAR_INPUT
{attribute 'ExpandFully'}
arr : ARRAY[0..5] OF INT;
END_VAR
```

Resulting **Properties** dialog box of frame in `visu_main`:



Attribute global_init_slot

Overview

The pragma `{attribute 'global_init_slot'}` defines the sequence of initialization of POUs or global variable lists.

Variables in a list (GVL or POU) are initialized from top to bottom.

If there are several global variable lists available, then the sequence of initialization is not defined.

The sequence of initialization is irrelevant for literal values, such as `1`, `'hello'`, `3.6`, or for constants of base data types. However, if there are dependencies between the different lists, you must define the sequence of initialization by yourself. To achieve this, you can assign a defined initialization slot to a GVL or a POU using the `attribute global_init_slot`.

Constants are initialized before the variables and in the same order as the variables. During initialization, the POUs are sorted according to the value for `<slot>`. Then the code for initializing the constants is generated. After that, the code for initializing the variables is generated.

Syntax

{attribute 'global_init_slot' := '<slot>'}

Replace <slot> by an integer value that defines the position in the initialization order. The default value for a POU (program, function block) is 50,000. The default value for a GVL is 49,990. A lower value provokes an earlier initialization. In case of several POUs or GVLs carrying the same value for the attribute `global_init_slot`, the sequence of their initialization remains undefined. This will be indicated as a detected programming error in the **Build** category of the **Messages** view (see Menu Commands, Online Help).

The pragma `{attribute 'global_init_slot'}` is valid for the entire GVL or POU and must therefore be located above the `VAR_GLOBAL` or POU declaration.

Example

The example includes 2 global variable lists *GVL_1* and *GVL_2* and a program `PLC_PRG` that uses variables from both lists.

GVL_1 uses the variable `B` for initializing a variable `A` which is initialized in the *GVL_2* with a value of `1000`.

GVL_1:
```
VAR_GLOBAL //49990
  A : INT := GVL_2.B*100;
END_VAR
```

GVL_2:
```
VAR_GLOBAL //49990
    B : INT := 1000;
  C : INT := 10;
END_VAR
```

PLC_PRG:
```
PROGRAM PLC_PRG //50000
VAR
    ivar: INT := GVL_1.A;
    ivar2: INT;
END_VAR
ivar:=ivar+1;
ivar2:=GVL_2.C;
```

When building this example, a programming error is issued in the **Build** category of the **Messages** view (see Menu Commands, Online Help) because `GVL_2.B` is used for initializing `GVL_1.A` before `GVL_2` has been initialized. To avoid this, use the `attribute global_init_slot` in order to position `GVL_2` before `GVL_1` in the sequence of initialization.

`GVL_2` must have a slot value of `49989` or lower to achieve the earliest initialization within the program.

GVL_2:
```
{attribute 'global_init_slot' := '100'}
VAR_GLOBAL
  B : INT := 1000;
  C : INT := 10;
END_VAR
```

You can even use `GVL_2.C` in the implementation part of `PLC_PRG` without a pragma because both GVLs are initialized before the program in either case.

Attribute hide

Overview

The pragma `{attribute 'hide'}` helps you to prevent variables or even whole signatures from being displayed within the functionality of listing components, page 634 or the input assistant or the declaration part in online mode or the symbol configuration. Only the variable subsequent to the pragma will be hidden.

If you know the instance path to a variable, you can address it in the code even if the `{attribute 'hide'}` is assigned.

This may be useful for library developers.

Syntax

```
{attribute 'hide'}
```

To hide all local variables of a signature, use the attribute hide_all_locals, page 610.

Example of a Hidden Variable

The function block *myPOU* is implemented using the attribute:

```
FUNCTION_BLOCK myPOU
VAR_INPUT
a:INT;
{attribute 'hide'}
a_invisible: BOOL;
a_visible: BOOL;
END_VAR
VAR_OUTPUT
b:INT;
END_VAR
```

In the main program 2 instances of function block *myPOU* are defined:

```
PROGRAM PLC_PRG
VAR
POU1, POU2: myPOU;
END_VAR
```

When assigning an input value to `POU1`, the functionality of listing components, page 634 that works on typing `POU1` in the implementation part of `PLC_PRG` will display the input variables `a` and `a_visible` (and the output variable `b`). The hidden input variable `a_invisible` will not be displayed.

Example of a Hidden Library POU

To hide the subordinate POUs (such as methods and properties) in the same way, insert the pragma `{attribute 'hide'}` also at the beginning of the declarations.

```
{attribute 'hide'}
FUNCTION_BLOCK FB_A
VAR_INPUT
END_VAR
VAR_OUTPUT
END_VAR
VAR
    iA : INT;
    iCount : INT;
    iInvisible : INT;
END_VAR

{attribute 'hide'}
METHOD METH_Count : INT
VAR_INPUT
END_VAR
iCount := iCount + 1;

{attribute 'hide'}
METHOD METH_Invisible : BOOL
VAR_INPUT
END_VAR
iInvisible := iInvisible + 1;

{attribute 'hide'}
PROPERTY PUBLIC prop_iA : INT
```

This has the effect that the POUs are not displayed during application development. Nevertheless, they are accessible by entering the instance path.

```
PROGRAM PLC_PRG
VAR
        fbHidden : FB_A; // Hidden function block from
library HiddenFunctionality
        iCounter : INT;
END_VAR
fbHidden.METH_Invisible();
iCounter := fbHidden.iInvisible;
```

Monitoring is not performed in online mode.

Attribute hide_all_locals

Overview

The pragma `{attribute 'hide_all_locals'}` helps you to prevent all local variables of a signature from being displayed within the functionality of listing components, page 634 or the input assistant or the declaration part in online mode or the symbol configuration. This may be useful for library developers.

This attribute is identical to assigning the attribute hide, page 609 to each particular of the local variables.

Syntax

```
{attribute 'hide_all_locals'}
```

Example

The function block *myPOU* is implemented using the attribute:

```
{attribute 'hide_all_locals'}
FUNCTION_BLOCK myPOU
VAR_INPUT
a:INT;
END_VAR
VAR_OUTPUT
b:BOOL;
END_VAR
VAR
c,d:INT;
END_VAR
```

In the main program 2 instances of function block *myPOU* are defined:

```
PROGRAM PLC_PRG
VAR
POU1, POU2: myPOU;
END_VAR
```

When assigning an input value to `POU1`, the functionality of listing components, page 634 that works on typing `POU1` in the implementation part of `PLC_PRG` will display the variables `a` and `b`. The hidden local variables `c` or `d` will not be displayed.

Attribute initialize_on_call

Overview

You can add the pragma *{attribute initialize_on_call}* to input variables. An input of a function block with this attribute will be initialized at any call of the function block. If an input expects a pointer and if this pointer has been removed due to an online change, then the input will be set to NULL.

Syntax

{attribute 'initialize_on_call'}

Attribute init_namespace

Overview

A variable of type STRING or WSTRING, which is declared with the pragma *{attribute init_namespace}* in a library, will be initialized with the current namespace of that library. For further information, refer to the description of the library management (see Functions and Libraries User Guide).

Syntax

{attribute 'init_namespace'}

Example

The function block `POU` is provided with all necessary attributes:

```
FUNCTION_BLOCK POU
VAR_OUTPUT
{attribute 'init_namespace'}
myStr: STRING;
END_VAR
```

Within the main program `PLC_PRG` an instance `fb` of the function block `POU` is defined:

```
PROGRAM PLC_PRG
VAR
fb:POU;
newString: STRING;
END_VAR
newString:=fb.myStr;
```

The variable *myStr* will be initialized with the current namespace, for example `MyLib.XY`. This value will be assigned to *newString* within the main program.

Attribute init_On_Onlchange

Overview

To initialize variables on online change, proceed as follows:

Step	Action
1	Attach the pragma *{attribute 'init_on_onlchange'}* to a variable to initialize this variable with each online change, page 243.
2	Open the **Build** tab of the **Properties** dialog box of the application (see Menu Commands, Online Help) and enter the string `no_fast_online_change` in the **Compiler defines** box.

Syntax

```
{attribute 'init_on_onlchange' }
```

Insert Location

Insert this pragma in the line above the declaration of the variables.

Attribute instance-path

Overview

You can add the pragma `{attribute instance-path}` to a local string variable. This local string variable will be initialized with the **Applications tree** path of the POU to which this string variable belongs. Applying this pragma presumes the use of the attribute reflection, page 630 for the corresponding POU and the additional attribute noinit, page 623 for the string variable.

Syntax

```
{attribute 'instance-path'}
```

Example

Assume the following function block *POU* being equipped with the attribute *'reflection'*:

```
{attribute 'reflection'}
FUNCTION_BLOCK POU
VAR
{attribute 'instance-path'}
{attribute 'noinit'}
str: STRING;
END_VAR
```

In the main program `PLC_PRG` an instance `myPOU` of function block *POU* is called:

```
PROGRAM PLC_PRG
VAR
myPOU:POU;
myString: STRING;
END_VAR
myPOU();
myString:=myPOU.str;
```

After initialization of instance *myPOU*, the string variable `str` gets assigned the path of instance *myPOU*, for example: *PLC.Application.PLC_PRG.myPOU*. This path will be assigned to variable *myString* within the main program.

NOTE: The length of a string variable may be arbitrarily defined (even >255). However, the string will be cut (from its back end) if it gets assigned to a string variable of a shorter length.

Attributes io_function_block, io_function_block_mapping

Overview

The `io_function_block` attribute marks a function block to allow the assignment to a channel in the I/O mapping of the device configuration. The function block is displayed in the **Select function block** dialog box.

The `io_function_block_mapping` attribute marks a parameter of the function block for usage when mapping the function block to a device channel. You can assign this attribute to multiple parameters of the function block. Automatic I/O mapping assigns the first parameter that matches the channel type (input, output, data type).

Syntax

```
{attribute 'io_function_block'}
{attribute 'io_function_block_mapping'}
```

Insert Location

Insert these parameters in the line above the first line in the declaration part of the function block, or in the line above the parameter declaration.

Example

```
{attribute 'io_function_block'}
FUNCTION_BLOCK Scale_Output_Int
VAR_INPUT
    iInput : INT;
    iNumerator : INT;
```

```
    iDenominator : INT :=1;
    iOffset : INT := 0;
END_VAR
VAR_OUTPUT
 {attribute 'io_function_block_mapping'}
    iOutput : INT;
END_VAR
VAR
```

Attribute is_connected

Overview

The `is_connected` attribute marks a Boolean variable of a function block to provide the information when the function block instance is called about whether the associated input of the POU has an assignment.

As a prerequisite, the `reflection` attribute, page 630 must be assigned to the respective function block.

Syntax

```
{attribute 'is_connected' := '<input variable>'}
```

Example

In the function block `FB`, a local variable is declared for each input variable (`in1` and `in2`). Each input variable is prefixed by the attribute `is connected` and the name of the input variable. The function block is assigned the pragma attribute `reflection`.

If an instance of the function block is called, the local variable is set to TRUE provided that the assigned input receives an assignment.

```
{attribute 'reflection'}
FUNCTION_BLOCK FB
VAR_INPUT
    in1: INT;
    in2: INT;
END_VAR
VAR
    {attribute 'is_connected' := 'in1'}
    in1_connection_info: BOOL;
    {attribute 'is_connected' := 'in2'}
    in2_connection_info: BOOL;
END_VAR
```

Assumption: When the function block instance is called, `in1` receives an external assignment and `in2` does not receive an assignment. This results in the following code:

```
in1_connection_info := TRUE;
in2_connection_info := FALSE;
```

Attribute linkalways

Overview

Use the pragma *{attribute 'linkalways'}* to mark a POU or a library POU for the compiler in a way so that they are always included into the compile information. During the build, objects with this option are compiled and become part of the application code. With a download, the POUs are downloaded to the controller.

The POU can be valid for the project (if located in the **POUs** tree) or for the application (if located in the **Devices** tree).

The compiler option **Link always** has the same effect.

Syntax

{attribute 'linkalways'}

Insert the pragma in the fist line of the declaration part of a POU or a library POU.

When you use the symbol configuration editor, the marked POUs are used as a basis for the selectable variables for the symbol configuration.

Example

The global variable list *MoreSymbols* GVL is implemented making use of the attribute *'linkalways'*. The variables declared are included in the application code, no matter whether they are accessed or not.

MoreSymbols GVL

```
{attribute 'linkalways'}
VAR_GLOBAL
        g_iAlpha: INT;
        g_iBravo: INT;
        g_iCharlie: INT;
END_VAR
```

As a result, the symbols of *MoreSymbols* GVL are available in the **Symbol configuration**.

Attribute monitoring

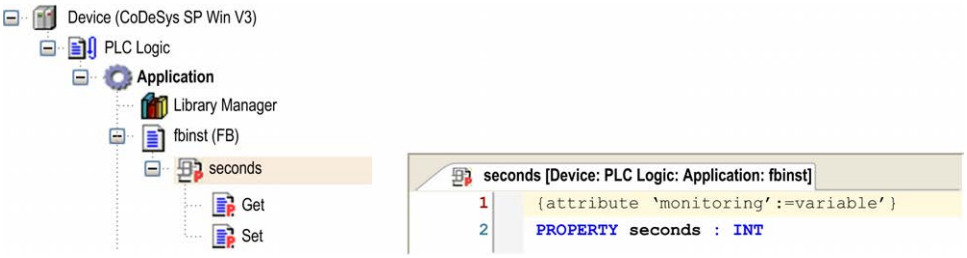
Overview

This attribute pragma allows you to get properties and function call results monitored in the online view of the IEC editor or in a watch list.

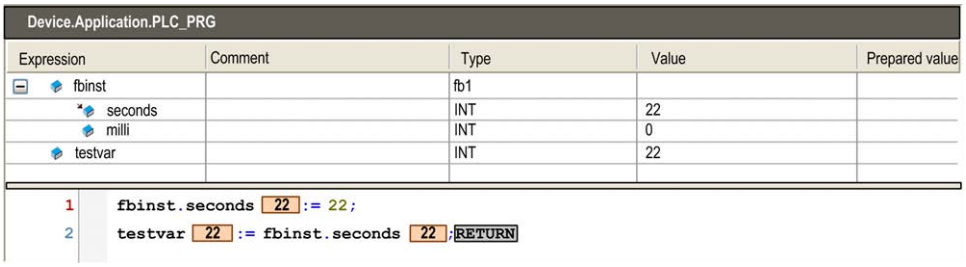
Monitoring of Properties

Add the pragma in the line above the property definition. Then the name, type, and value of the variables of the property will be displayed in the online view of the POU using the property or in a watch list. Therein, you can also enter prepared values to force variables belonging to the property.

Example of property prepared for variable monitoring



Example of monitoring view



Monitoring the Current Value of the Property Variables

There are two different ways to monitor the current value of the property variables. For the particular use case, consider carefully which attribute is suitable to actually get the desired value. This will depend on whether operations on the variables are implemented within the property:

1. Pragma {attribute 'monitoring':='variable'}

An implicit variable is created for the property, which will get the current property value whenever the application calls the set or get method. The latest value stored in this implicit variable will be monitored.

Syntax

{attribute 'monitoring':='variable'}

2. Pragma {attribute 'monitoring':='call'}

You can only use this attribute for properties returning simple data types or pointers, not for structured types.

The value to be monitored is read or written by a direct call of property: the monitoring service of the runtime system executes the `Get` or `Set` method of the property function including the implementation part of the property.

> **NOTE:** When choosing this monitoring type instead of using an intermediate variable (see *1. Pragma*), consider possible side effects due to any operations implemented within the property.
>
> **NOTE:** The *monitoring* pragma is also evaluated by the symbol configuration, page 508. If the value *variable* was specified, only a read-access on the property is available in the symbol configuration.

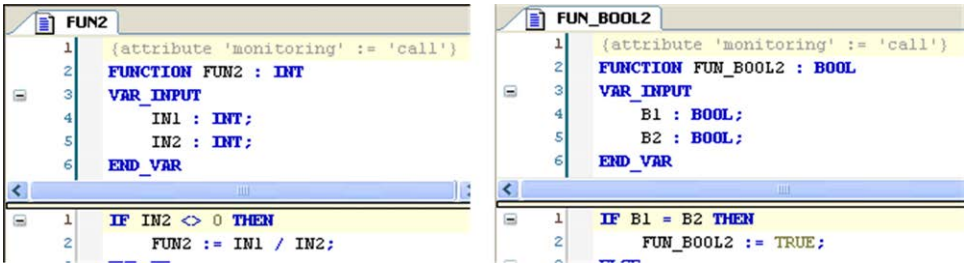
Syntax

{attribute 'monitoring':='call'}

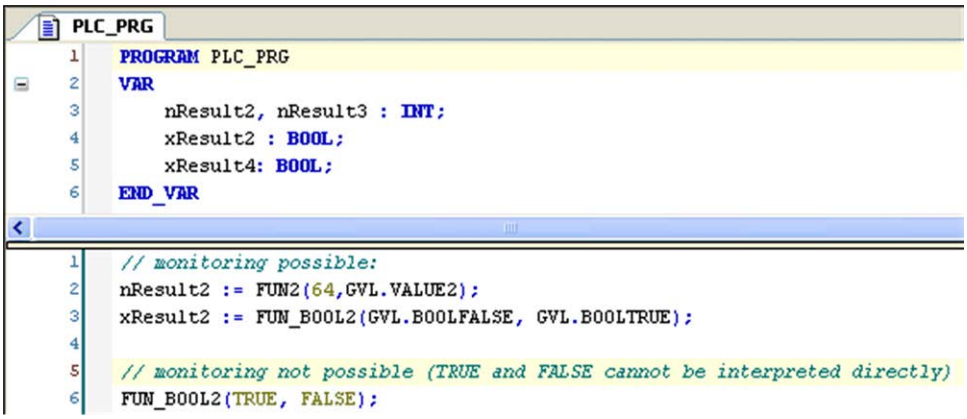
Monitoring of Function Call Results

You can use function call monitoring for any constant value that can be interpreted as 4 byte numerical value (for example, INT, SHORT, LONG). For the other input parameters (for example, BOOL), use a variable instead of a constant parameter. Add the pragma `{attribute 'monitoring':='call'}` in the line above the function declaration. You can then monitor this variable in the text editor view in online view of the POU in which a variable gets assigned the result of a function call. You can also add the variable to a watch list for the same purpose. To get the variable immediately provided within a watch view, execute the command **Add watchlist**.

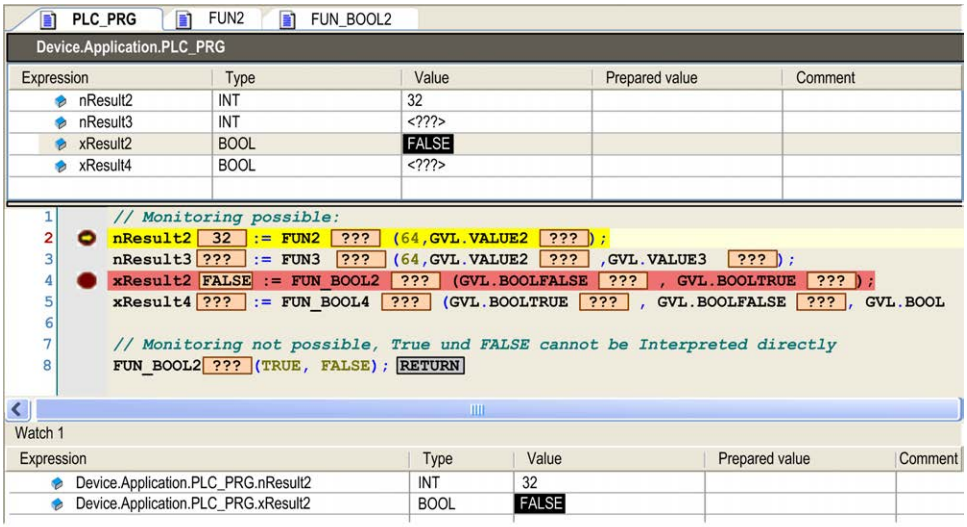
Example 1: Functions `FUN2` and `FUN_BOOL2` with attribute `'monitoring'`



Example 2: Call of functions `FUN2` and `FUN_BOOL2` in a program POU



Example 3: Function calls in online mode:



Monitoring of Variables with an Implicit Call of an External Function

For monitoring variables with an implicit call of an external function, the following conditions have to be fulfilled:

- The function is marked with `{attribute 'monitoring' := 'call'}`.
- The function is marked as **Link Always**.
- The variable is marked with `{attribute 'monitoring_instead' := 'MyExternalFunction(a,b,c)'}`.
- The values `a,b,c` are integer values and match the input parameters of the function to call.

  NOTE: Forcing or writing of functions is not supported. You can implicitly implement forcing by adding an additional input parameter for the particular function that serves as an internal force flag.

  NOTE: Function monitoring is not possible on the compact runtime system.

Attribute monitoring_display

Overview

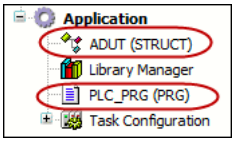
Insert this attribute pragma in the declaration part of a function block or a structure to monitor the value of the specified member (property or variable) in the online view. The value is displayed in the first line of monitored variables of the respective function block or structure.

Syntax

```
{attribute 'monitoring_display' := '<component name>'}
```

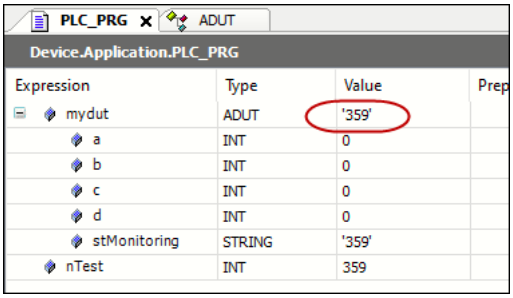
Insert the pragma as the first line of the declaration part of a function block or a structure.

Example




```
{attribute 'monitoring_display' := 'stMonitoring'}
TYPE ADUT :
STRUCT
 val1: INT;
 val2: INT;
 stMonitoring: STRING := 'to be monitored';
END_STRUCT
END_TYPE

PROGRAM PLC_PRG
VAR
 mydut: ADUT;
 nTest: INT;
END_VAR
nTest:= iCounter + 1;
mydut.stMonitoring := INT_TO_STRING(nTest);
```



Attribute monitoring_encoding

Overview

Add the pragma `Attribute monitoring_encoding` to string variables and aliases. Values of variables marked with this pragma are decoded in UTF-8 format, page 642 during monitoring.

Syntax

```
{attribute 'monitoring_encoding' := 'UTF8'}
```

Example

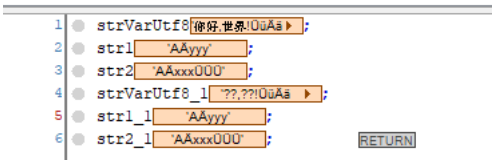
Insert the pragma `Attribute monitoring_encoding` above the variable declaration.

```
PROGRAM PLC_PRG
VAR
    {attribute 'monitoring.encoding' := 'UTF8'}
    strDat : STRING := 'abc';
    attribute 'monitoring_encoding' := 'UTF-8'}
    strVarUtf8: STRING := UTF8#'你好,世界!ÜüÄäÖö';
    {attribute 'monitoring_encoding' := 'UTF-8'}
    str1: STRING := UTF8#'AÄyyy';
    {attribute 'monitoring_encoding' := 'UTF-8'}
    str2: STRING := UTF8#'AÄxxxÜÜÜ';

    strVarUtf8_1: STRING := '你好,世界!ÜüÄäÖö';
    str1_1: STRING := 'AÄyyy';
    str2_1: STRING := 'AÄxxxÜÜÜ';
END_VAR
```

Device.Application.PLC_PRG

Expression	Type	Value
strVarUtf8	STRING	'你好,世界!ÜüÄäÖö'
str1	STRING	'AÄyyy'
str2	STRING	'AÄxxxÜÜÜ'
strVarUtf8_1	STRING	'??,??!ÜüÄäÖö'
str1_1	STRING	'AÄyyy'
str2_1	STRING	'AÄxxxÜÜÜ'



Attribute namespace

Overview

In combination with the attribute symbol, page 631, the pragma *{attribute namespace}* allows you to redefine the namespace of project variables. You can apply it on complete POUs, like GVLs or programs, but not on particular variables. The concerned variables will be exported with the new namespace definition to a symbol file and after a download of this file be available on the controller.

This also allows you to access variables from POUs or visualizations which originally have got different namespaces. For example, it allows you to run a previous EcoStruxure Machine Expert visualization also in a later EcoStruxure Machine Expert environment.

For further information, refer to the description of the symbol configuration. A new symbol file will be created at a download or online change of the project. It is downloaded to the controller together with the application.

Syntax

```
{attribute 'namespace' := '<namespace>'}
```

Example of a Namespace Replacement for the Variables of a Program

```
{attribute 'namespace':='prog'}
PROGRAM PLC_PRG
VAR
  {attribute 'symbol' := 'readwrite'}
  iVar:INT;
  bVar:BOOL;
END_VAR
```

If *iVar*, for example, was accessed by *App1.PLC_PRG.ivar* before, now it is accessible via *prog.ivar*.

Further Replacement Examples

Original Namespace	Variable	Namespace Replacement	Access on the Variable Within the Current Project
App1.Lib2.GVL2	Var07	{attribute 'namespace':=''}	.Var07
App1.GVL2	Var02	{attribute 'namespace':='Ext''}	Ext.Var02
App1.GVL2.FB1	Var02	{attribute 'namespace':='App1.GVL2''}	App1.GVL2.Var02

The replacements shown in the table result in the following entries in the symbol file:

```
<NodeList>
 <Node name="">
     <Node name="Var07" type="T_INT" access="ReadWrite">
 </Node>
</NodeList>
<NodeList>
 <Node name="Ext">
  <Node name="Var02 " type="T_INT" access="ReadWrite"></
Node>
 </Node>
</NodeList>
<NodeList>
 <Node name="App1">
  <Node name="GVL2">
   <Node name="Var02 " type="T_INT" access="ReadWrite"></
Node>
     </Node>
 </Node>
</NodeList>
```

Attributes no_assign, no_assign_warning

Overview

Insert the pragma *{attribute 'no assign'}* as first line of the declaration part of a function block. This has the effect that compile **errors** in the **Messages** view (see Menu Commands, Online Help) are generated if an instance of the function block is assigned to another instance of the same function block. For example, you might want to avoid such assignments if the function block contains pointers. This could cause issues because they are copied when values are assigned.

> **NOTE:** Use the *{attribute 'no assign'}* in function blocks with internal pointers to help to avoid assigning an instance of the function block to another instance of the same function block.

The pragma `'no_assign_warning'` has the same functions as the pragma *'no assign'* but issues detected alerts as category **warnings** in the **Messages** view (see Menu Commands, Online Help).

Assignment of Function Block Instances Containing Pointers

In this example, the value assignment of function block instances causes issues when `fb_exit` is executed:

```
VAR_GLOBAL
inst1 : TestFB;
    awsBufferLogFile : ARRAY [0..9] OF WSTRING(66);(* Area:
0, Offset: 0x1304 (4868)*)
    LogFile : SEDL.LogRecord := (sFileName := 'LogFile.log',
pBuffer := ADR(awsBufferLogFile), udiMaxEntriesFile :=
UDINT#10000, udiMaxBuffered := UDINT#10, uiLineSize :=
UINT#64, wsSep := " ", xCircular := TRUE, siDateFormat :=
SINT#0, siTimeFormat := SINT#0);
END_VAR
PROGRAM PLC_PRG
VAR
    inst2 : TestFB := inst1;
    LogFileNew : LogRecord := LogFile;
END_VAR
```

In this case, `LogRecord` manages a list of pointers. Different actions are executed for them if `fb_exit` applies. When assigning function block instances, `fb_exit` will be executed twice which causes an issue. Prevent this by adding the `no_assign` attribute to the declaration of function block `TestFB`:

```
{attribute 'no_assign'}
FUNCTION_BLOCK TestFB
VAR_INPUT
...
```

The following compile errors are reported when the pragma *{attribute 'no assign'}* is used:

```
C0328: Assignment not allowed for type TestFB
C0328: Assignment not allowed for type LogRecord
```

If the pragma *no assign_warning* is used for the function block `TestFB`, the `C0328` messages are also issued, but as category **warning** in the **Messages** view (see Menu Commands, Online Help).

Attribute no_check

Overview

You can add the pragma *{attribute 'no_check'}* to the declaration of a POU in order to suppress the call of any POUs for implicit checks. As checking functions may influence the performance, apply this attribute to POUs that are frequently called or already approved.

> **NOTE:** This attribute has an automatic effect also on the child objects of a POU.

Example: In programs with this attribute, no check functions are executed. Not even for actions that are assigned to this program.

Syntax

{attribute 'no_check'}

Attribute no_copy

Overview

Generally, an online change will require a reallocation of instances, for example of POUs. The value of the variables within this instance will get copied.

If, however, the pragma *{attribute no_copy}* is added to a variable, an online change copy of this variable will not be performed; this variable will be initialized instead. This can be reasonable in case of local pointer variable, pointing on a variable actually shifted by the online change (and thus having a modified address).

Syntax

```
{attribute 'no_copy'}
```

Attribute no-exit

Overview

If a function block provides an *FB_exit* method, page 579 , you can suppress its call for a special instance with the help of assigning the pragma *{attribute no-exit}* to the function block instance.

Syntax

```
{attribute 'no-exit'}
```

Example

Assume the method *FB_Exit* being added to a function block named POU:



In the main program *PLC_PRG*, 2 variables of type POU are instantiated:

```
PROGRAM PLC_PRG
VAR
POU1 : POU;
{attribute 'no-exit'}
POU2 : POU;
END_VAR
```

When variable *bInCopyCode* becomes TRUE within *POU1*, the method *FB_Exit* is called exiting an instance that will get copied afterwards (online change). The method *FB_Exit* is not called in context of the function block instance *POU2*.

Attribute no_init

Overview

Variables provided with the pragma *{attribute no_init}* will not be initialized implicitly. The pragma belongs to the variable declared subsequently.

Syntax

```
{attribute 'no_init'}
```

also possible

```
{attribute 'no-init'}
{attribute 'noinit'}
```

Example

```
PROGRAM PLC_PRG
VAR
A : INT;
{attribute 'no_init'}
B : INT;
END_VAR
```

If a reset is performed on the associated application, the integer variable `A` will be again initialized implicitly with 0, whereas variable `B` maintains the value it is currently assigned to.

Attribute no_instance_in_retain

Overview

The pragma *{attribute 'no_instance_in_retain'}* helps to avoid that the instance of a certain function block instance is stored in the retain memory area.

Insert it as first line of the declaration of the function block. This has the effect that a message is created in case any instance of the POU is declared as a `RETAIN` variable.

Syntax

```
{attribute 'no_instance_in_retain'}
```

Attribute no_virtual_actions

Overview

This attribute is valid for function blocks, which are derived from a base function block implemented in SFC, and which are using the main SFC workflow of the base class. The actions called therein show the same virtual behavior as methods. Therefore, the base class actions may be overridden by specific implementations related to the derived classes.

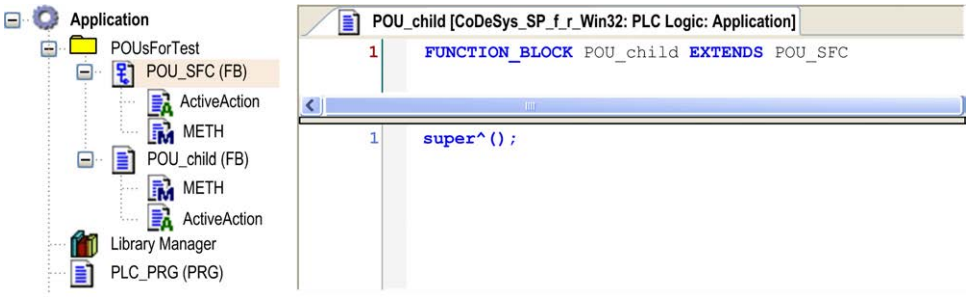
In order to help to prevent the action of the base class from being overridden, you can assign the pragma `{attribute 'no_virtual_actions'}` to the base class.

Syntax

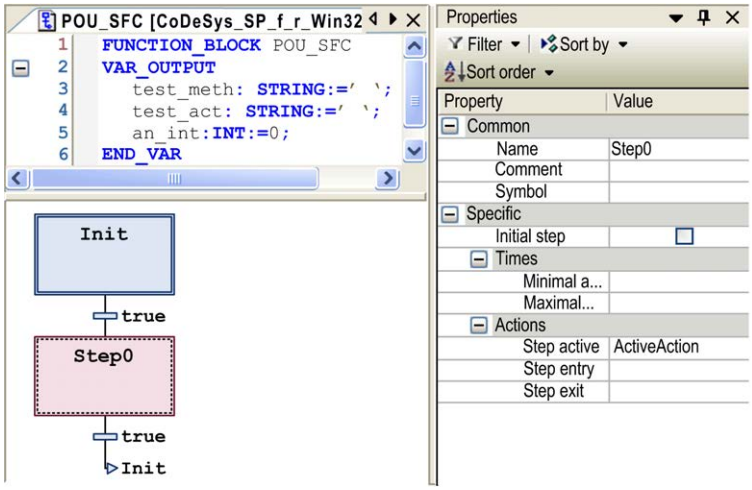
```
{attribute 'no_virtual_actions'}
```

Example

In the following example, the function block `POU_SFC` provides the base class to be extended by the function block `POU_child`.



By use of the keyword `SUPER`, the derived class `POU_child` calls the workflow of the base class that is implemented in SFC.



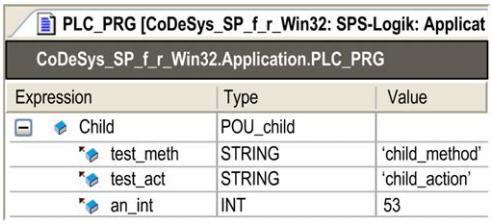
The exemplary implementation of this workflow is restricted to the initial step. This is followed by 1 single step with associated step action *ActiveAction* concerned with the assignment of the output variables:

```
an_int:=an_int+1;           // counting the action calls
test_act:='father_action'; // writing string variable test_
act
METH();                    // Calling method METH for writing
string variable test_meth
```

In case of the derived class `POU_child`, the step action will be overwritten by a specific implementation of *ActiveAction*. It differs from the original one by assigning the string `'child_action'` instead of `'father_action'` to variable `test_act`.

Likewise, the method `METH`, assigning the string `'father_method'` to variable `test_meth` within the base class, will be overwritten such that `test_meth` will be assigned to `'child_method'` instead.

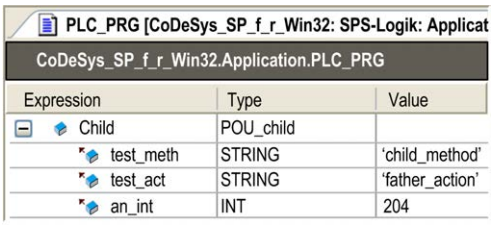
The main program `PLC_PRG` will execute repeated calls to *Child* (an instance of `POU_child`). As expected, the actual value of the output string report the call to action and method of the derived class:



You can observe a different behavior if the base class is preceded by the attribute `'no_virtual_actions'`

```
{attribute 'no_virtual_actions'}
FUNCTION_BLOCK POU_SFC...
```

Whereas method `METH` will still be overwritten by its implementation within the derived class, a call of the step action will now result in a call of action *ActiveAction* of the base class. Therefore, `test_act` will be assigned to string `'father_action'`.



Attribute pingroup

Overview

Insert the pragma *{attribute 'pingroup' := '<groupname>'}* in the declaration of a function block, for grouping the input pins or output pins (parameters). Then, in the respective box of the FBD and LD editor, each pin group can be displayed, folded, or unfolded. Multiple groups are possible and are differentiated by their names. The particular state (folded/unfolded) per box is saved in the project options.

Inputs and outputs without attribute `pingroup` are always displayed above any possible group or groups.

Syntax

```
{attribute 'pingroup' := '<groupname>'}
```

Example

Two groups are defined:

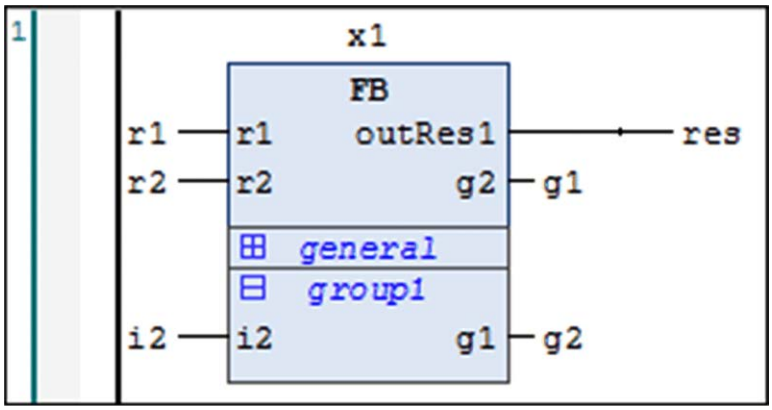
- general `(i1, out1)`
- group1 `(i2, g1)`

`r1, r2, outRes1` and `g2` are always displayed.

```
FUNCTION_BLOCK FB
VAR_INPUT
   r1 : REAL;
   {attribute 'pingroup' := 'general'}
   i1 : INT;
   {attribute 'pingroup' := 'group1'}
```

```
   i2 : INT;
   r2 : REAL;
END_VAR
VAR_OUTPUT
   outRes1 : REAL;
   {attribute 'pingroup' := 'general'}
   out1 : INT;
   {attribute 'pingroup' := 'group1'}
   g1 : INT;
   g2 : REAL;
END_VAR
```

Pingroups in FBD editor



Attribute pin_presentation_order_inputs/outputs

Overview

The pragmas define the order in which the inputs and outputs of a function block are displayed in graphical language editors.

Syntax

```
{attribute 'pin_presentation_order_inputs' :=
'<input_k>,<input_l>,*,<input_m>'}
{attribute 'pin_presentation_order_outputs' :=
'<output_k>,<output_l>,*,<output_m>'}
```

The * character is the separator between the beginning and the end of the sorted list of input or output parameters. The separator is replaced by undefined input or output parameters. If the separator is not available, then the input or output parameters that are not defined explicitly in the pragma will be added to the end of the sorted list.

The pragmas are inserted in the first line in the declaration part of a function block.

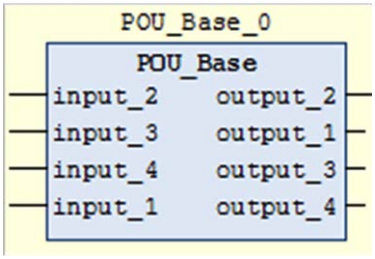
NOTE: The pragmas *attribute 'pin_presentation_order_inputs* and *attribute 'pin_presentation_order_outputs* are not evaluated when the pragma *pingroup* is used.

Example

```
{attribute 'pin_presentation_order_inputs' :=
'input_2,*,input_1'}
{attribute 'pin_presentation_order_outputs' :=
'output_2, output_1}
FUNCTION_BLOCK POU_BASE
VAR_INPUT
    input_1 : BOOL;
```

```
    input_2 : INT;
    input_3 : INT;
    input_4 : INT;
END_VAR
VAR_OUTPUT
    output_1 : BOOL;
    output_2 : INT;
    output_3 : INT;
    output_4 : BOOL;
END_VAR
```

This sample pragma definition leads to the following order of the input and output pins of the *POU_Base* function block:



Attribute obsolete

Overview

You can add an obsolete pragma to a data type definition in order to cause a user-defined alert during a build, if the respective data type (structure, function block, and so on) is used within the project. Thus, you can announce that the data type is not used any longer.

Unlike a locally used message pragma, page 585, this alert is defined within the definition and thus global for all instances of the data type.

This pragma instruction is valid for the current line or - if placed in a separate line - for the subsequent line.

Syntax

{attribute 'obsolete' := 'user-defined text'}

Example

The obsolete pragma is inserted in the definition of function block `fb1`:

```
{attribute 'obsolete' := 'datatype fb1 not valid!'}
FUNCTION_BLOCK fb1
VAR_INPUT
i:INT;
END_VAR
...
```

If `fb1` is used as a data type in a declaration, for example, `fbinst: fb1;` the following alert will be dumped when the project is built:

```
'datatype fb1 not valid'
```

Attribute pack_mode

Overview

The pragma `{attribute 'pack_mode'}` defines the mode a data structure is packed while being allocated. Set the attribute on top of a data structure. It influences the packing of the whole structure.

NOTE: Do not select the option **Compatibility Layout** in the **Symbol Configuration** editor, page 509 if the pragma `{attribute 'pack_mode'}` is used in the programming code.

Syntax

```
{attribute 'pack_mode' := '<value>'}
```

The placeholder `<value>` can have the following values:

'pack_mode'	Associated packing method
0	Aligned
1	1-byte-aligned
2	2-byte-aligned
4	4-byte-aligned
8	8-byte-aligned

Depending on the structure, there may be no difference in the memory mapping of the individual modes. For example, the memory allocation of a structure with `pack_mode = 4` can correspond to that of `pack_mode = 8`.

If structures are combined to arrays, bytes are added at the end of each structure to achieve that the next structure is aligned.

NOTE: We refer to “natural” alignment as when `'pack_mode'` is not used for a structure or corresponds to the alignment of the processor of the controller.

On some controllers, like the Modicon M262 Logic/Motion Controllers and Modicon M241 Logic Controllers, using a non natural alignment for structures may lead to an exception when accessing elements of such structures that are unaligned (for example, a 4-byte variable located on an address not divisible by 4), with a POINTER variable.

⚠WARNING

UNINTENDED EQUIPMENT OPERATION

Use “naturally” aligned variables for Modicon M262 and M241 controllers.

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

Furthermore, do not expose non-naturally aligned structures from libraries.

Do not use non-naturally aligned structures inside of visualization frames. Instead copy those structures into naturally aligned structures, and pass those to visualization frames.

Example

```
{attribute 'pack_mode' := '1'}
TYPE myStruct:
STRUCT
 Enable: BOOL;
 Counter: INT;
 MaxSize: BOOL;
 MaxSizeReached: BOOL;
END_STRUCT
END_TYPE
```

A variable of data type `myStruct` is instantiated aligned.

If the address of its component `Enable` is `0x0100`, then the component `Counter` will follow on address `0x0101`, `MaxSize` on `0x0103` and `MaxSizeReached` on `0x0104`.

With `pack_mode=2`, `Counter` would be found on `0x0102`, `MaxSize` on `0x0104` and `MaxSizeReached` on `0x0106`.

Attribute ProcessValue

Overview

The `ProcessValue` attribute marks a component of a structure. In the CFC editor, you can execute the **Use attributed member as input** command (see Menu Commands, Online Help) to connect this structure to a scalar type input.

Syntax

```
{attribute 'ProcessValue'}
```

Insert Location

Insert the `ProcessValue` attribute in the line above the structure variable.

Example

```
TYPE QINT :
STRUCT
    Status : STRING;
    {attribute 'ProcessValue'}
    Value1 : INT;
    Value2 : INT;
END_STRUCT
END_TYPE
```

Attribute qualified_only

Overview

When the pragma `{attribute 'qualified_only'}` is assigned on top of a global variable list, the variables of this list can only be accessed by using the global variable name, for example `gvl.g_var`. This works even for variables of enumeration type. It can be useful to avoid name mismatch with local variables.

Syntax

{attribute 'qualified_only'}

Example

Assume the following global variable list (GVL) is provided with attribute `'qualified_only'`:

```
{attribute 'qualified_only'}
VAR_GLOBAL
iVar:INT;
END_VAR
```

Within POU `PLC_PRG`, the global variable has to be called with the prefix `GVL`, as shown in this example:

```
GVL.iVar:=5;
```

The following incomplete call of the variable will be detected as an error:

```
iVar:=5;
```

Attribute reflection

Overview

Attach the pragma `{attribute 'reflection'}` to POUs containing variables that are tagged with the *instance-path* attribute, page 612 or the *is-connected* attribute, page 614.

Syntax

{attribute 'reflection'}

Example

Refer to the `attribute instance-path` example, page 613 and to the *attribute is-connected* example, page 614.

Attribute subsequent

Overview

The pragma `{attribute 'subsequent'}` forces variables to be allocated in a row at one location in memory. If the list changes, the whole list will be allocated at a new location. This pragma is used in programs and global variable lists (GVL).

Syntax

{attribute 'subsequent'}

NOTE: If one variable in the list is `RETAIN`, the whole list will be located in retain memory.

NOTE: `VAR_TEMP` in a program with attribute `subsequent` will be detected as a compiler error.

Attribute suppress_warning

Overview

The pragma `attribute 'suppress_warning'` has the effect that the compiler advisories specified in the definition of this pragma are suppressed.

Syntax

{attribute 'suppress_warning' := '<warning id1>',"<warning id2>',...}

Insert the `Attribute suppress_warning` in the line above the declaration part of a POU or DUT.

Example

```
{attribute 'suppress_warning' := '0125'}
TYPE DUT_1 :
(
    enum_member1 := 0,
    enum_member2 := 0
);
END_TYPE
```

Result: The compiler message C0125 is suppressed.

Attribute symbol

Overview

The pragma `{attribute 'symbol'}` defines which variables are to be handled in the symbol configuration.

The following export operations are performed on the variables:

- Variables are exposed as symbols in the symbol configuration.
- Variables are exported to an XML file in the project directory.
- Variables are exported to a file not visible and available on the target system for external access, for example, by an OPC server.

Variables provided with that attribute will be downloaded to the controller even if they have not been configured or are not visible within the symbol configuration editor.

> **NOTE:** The symbol configuration has to be available as an object below the respective application in the **Tools Tree**.

Additionally you can add the pragma `{attribute 'symbol'}` in front of a function block type to expose all instances of that function block type, except function block instances created as array elements, to the symbol configuration. To exclude a dedicated member variable of the function block from being exposed, insert `{attribute 'symbol':='none'}` in front of this variable.

> **NOTE:** It is a good practice to apply the attribute to each function block instance individually. Insert it in front of a function block type only if you are sure that *all* function block instances of this type are to be exposed.

Syntax

{attribute 'symbol' := 'none' | 'read' | 'write' | 'readwrite'}

Access is only allowed on symbols coming from programs or global variable lists. For accessing a symbol, specify the symbol name completely.

You can assign the pragma definition to particular variables or collectively to all variables declared in a program.

- To be valid for a single variable, place the pragma in the line before the variable declaration.
- To be valid for all variables contained in the declaration part of a program, place the pragma in the first line of the declaration editor. In this case, you can also modify the settings for particular variables by explicitly adding a pragma.

The possible access on a symbol is defined by the following pragma parameters:

- `'none'`
- `'read'`
- `'write'`
- `'readwrite'`

If no parameter is defined, the default `'readwrite'` will be valid.

Example

With the following configuration, the variables `A` and `B` will be exported with read and write access. Variable `D` will be exported with read access.

```
{attribute 'symbol' := 'readwrite'}
PROGRAM PLC_PRG
VAR
A : INT;
B : INT;
{attribute 'symbol' := 'none'}
C : INT;
{attribute 'symbol' := 'read'}
D : INT;
END_VAR
```

Attribute to_string

Overview

The `to_string` attribute determines the output type of the result of an enumeration component conversion with the `TO_STRING` operator. If the enumeration declaration contains the pragma, the name of the enumeration component is displayed as string instead of the numeric value.

Syntax

```
{attribute 'to_string'}
```

Insert Location

Insert the `to_string` attribute in the first line above the declaration part of the enumeration.

Example

Declaration of the enumeration `color`:

```
{attribute 'to_string'}
TYPE color :
(
    red := 0,
    blue := 1,
    green := 2
);
END_TYPE
```

Conversion with `TO_STRING`:

```
PROGRAM PLC_PRG
VAR
    i_color: color;
    s_show_color: STRING;
END_VAR
i_color := 1;
s_show_color := TO_STRING(i_color);
```

In this case, `s_show_color` displays the value `'blue'` instead of `1` as the conversion result.

Attribute warning disable, warning restore

Overview

You can use the pragma `warning disable` to suppress advisories, page 631. To re-enable the display of the advisory, use the pragma `warning restore`. For a list of possible compiler advisories, open the **Project > Project Settings – Compiler Warnings** (refer to the Menu Commands Online Help).

Syntax

{warning disable <compiler ID>}

Compiler ID: Every advisory and every error detected by the compiler has a unique ID, which is displayed at the beginning of the description.

Example Compiler Messages

```
------ Build started: Application: Device.Application ------
typify code ...
C0195: Implicit conversion from signed Type 'SINT' to
unsigned Type 'UINT' : possible change of sign
Compile complete -- 0 errors
```

Example

```
VAR
    {warning disable C0195}
    test1 : UINT := -1;
    {warning restore C0195}
    test2 : UINT := -1;
END_VAR
```

In this example, an advisory will be detected for `test2`. But no advisory will be detected for `test1`.

The Smart Coding Functionality

Smart Coding

Overview

Wherever identifiers (like variables or function block instances) can be entered (this can be inside of the IEC 61131-3 language editors or inside **Watch, Trace, Visualization** windows), the smart coding functionality is available. You can customize this feature in the **SmartCoding** section of the **Tools > Options** dialog box.

Support in Identifier Insertion

The smart coding functionality helps to insert a correct identifier:

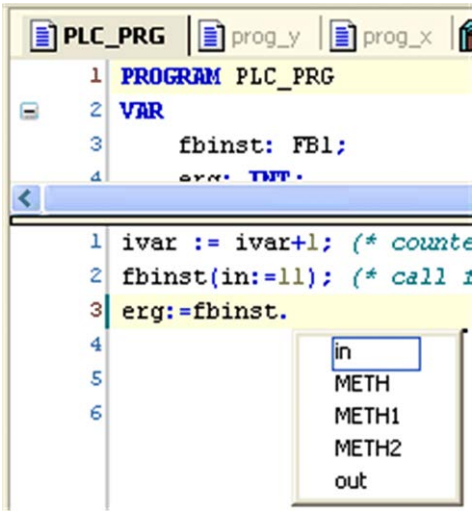
- If you - at any place, where a global identifier can be inserted - insert a dot (.) instead of the identifier, a selection box will display. It lists the currently available global variables. You can choose one of these elements and press the RETURN key to insert it behind the dot. You can also insert the element by double-clicking the list entry.
- If you enter a function block instance or a structure variable followed by a dot (.), then a selection box will appear. It lists the input and output variables of the corresponding function block or the structure components. You can choose the desired element by pressing the RETURN key or by double-clicking the list entry to insert it.
- In the ST editor, if you enter any string and press CTRL+SPACE, a selection box will display. It lists the POUs and global variables available in the project. The first list entry, which is starting with the given string, will be selected. Press the RETURN key to insert it into the program.

Examples

The smart coding functionality offers components of structure:



The smart coding functionality offers components of a function block:



Data Types

What's in This Chapter

General Information 635
Standard Data Types 635
Extensions to IEC Standard 644
User-Defined Data Types 650

General Information

Data Types

Overview

You can use standard data types, page 635, user-defined data types, page 650, or instances of function blocks when programming in EcoStruxure Machine Expert. Each identifier is assigned to a data type. This data type dictates how much memory space will be reserved and what type of values it stores.

Standard Data Types

Standard Data Types

Overview

EcoStruxure Machine Expert supports the data types, page 635 described by standard IEC61131-3.

The following elementary data types are described in this chapter:

- BOOL, page 635
- Integer, page 636
- REAL / LREAL, page 637
- STRING, page 638
- WSTRING, page 639
- Date and time data types, page 639
- ANY and ANY_<type>, page 640

Additionally, some standard-extending data types, page 644 are supported and you can define your own user-defined data types, page 650.

BOOL

BOOL type variables can have the values TRUE (1) and FALSE (0). 8 bits of memory space are reserved.

For further information, refer to the chapter *BOOL constants*, page 746.

NOTE: You can use implicit checks to validate the conversion of variable types (refer to the chapter *POUs for Implicit Checks*, page 191).

Integer

The table lists the available integer data types. Each of the types covers a different range of values. The following range limitations apply.

Data Type	Lower Limit	Upper Limit	Memory Space
BYTE	0	255	8 bit
WORD	0	65,535	16 bit
DWORD	0	4,294,967,295	32 bit
LWORD	0	2^{64}-1	64 bit
SINT	–128	127	8 bit
USINT	0	255	8 bit
INT	–32,768	32,767	16 bit
UINT	0	65,535	16 bit
DINT	–2,147,483,648	2,147,483,647	32 bit
UDINT	0	4,294,967,295	32 bit
LINT	-2^{63}	2^{63}-1	64 bit
ULINT	0	2^{64}-1	64 bit

NOTE: Conversions from larger types to smaller types may result in loss of information.

For further information, refer to the description of number constants, page 750.

NOTE: You can use implicit checks to validate the conversion of variable types (refer to the chapter *POUs for Implicit Checks*, page 191).

Target-Independent Integer Data Types __UXINT, __XINT, __XWORD

To create IEC code suitable for the different target systems, use the target-independent integer data types __UXINT, __XINT, and __XWORD. The compiler converts these data types into the appropriate standard data types supported by the target system. Systems with address registers of 32-bit and 64-bit width are supported.

The following type conversion operators are provided:

Target-independent data types	On 64-bit platforms	On 32-bit platforms
__UXINT	ULINT	UDINT
__XINT	LINT	DINT
__XWORD	LWORD	DWORD

REAL / LREAL

The data types REAL and LREAL are floating-point types. They represent rational numbers.

Characteristics of REAL and LREAL data types:

Data type	Lower limit	Upper limit	Memory Space	Lowest consecutive integer	Highest consecutive integer
REAL	-3.402823e+38	3.402823e+38	32-bit	-16777216	16777216
LREAL	-1.7976931348623158e +308	1.7976931348623158e +308	64-bit	-9007199254740992	9007199254740992

NOTE:

The *lowest consecutive integer (counter)*, *x*, is the lowest integer for which all integers greater than or equal to *x* can be exactly represented but *x* - 1 cannot be represented in floating-point format.

The *highest consecutive integer (counter)*, *x*, is the highest integer for which all integers less than or equal to *x* can be exactly represented but *x* + 1 cannot be represented in floating-point format.

Example:

```
PROGRAM PLC_PRG
VAR
    rMax: REAL := 3.402823E+38; // Largest number
    rPosMin : REAL := 1.0E-44; // Smallest positive number
    rNegMax: REAL := -1.0E-44; // Largest negative number
    rMin: REAL := -3.402823E+38; // Smallest number
    lrMax: LREAL := 1.7976931348623158E+308; // Largest
number
    lrPosMin : LREAL := 4.94065645841247E-324; // Smallest
positve number
    lNegMax: LREAL := -4.94065645841247E-324; // Largest
negative number
    lrMin: LREAL := -1.7976931348623158E+308; // Smallest
number
END_VAR
```

NOTE: The support of data type LREAL depends on the target device. See in the corresponding documentation whether the 64-bit type LREAL gets converted to REAL during compilation (possibly with a loss of information) or persists.

NOTE: If a REAL or LREAL is converted to SINT, USINT, INT, UINT, DINT, UDINT, LINT, or ULINT and the value of the real number is out of the value range of that integer, the result will be undefined and will depend on the target system. Even an exception is possible in this case. In order to get target-independant code, handle any range exceedance by the application. If the REAL/LREAL number is within the integer value range, the conversion will work on all systems in the same way.

When assigning `i1 := r1;` an error is detected. Therefore, the previous note applies when using conversion operators, page 706 such as the following:

```
i1 := REAL_TO_INT(r1);
```

For further information, refer to REAL/LREAL constants (operands), page 751.

NOTE: You can use implicit checks to validate the conversion of variable types (refer to the chapter *POUs for Implicit Checks*, page 191).

STRING

The STRING data type can be encoded in Latin 1 or in UTF-8 format. For project-wide interpretation as UTF-8, enable the parameter **UTF8 Encoding for STRING** in the **Project Settings > Compile options** dialog box. For further information, refer to Project-Wide UTF-8 Encoding, page 643.

> **NOTE:** Before you set the encoding format to UTF-8, execute the static analysis rule SA0175 (refer to the Menu Commands Online Help for details) on your code to help detect constructs that may cause issues with UTF-8 encoding.
>
> **NOTE:** If you use UTF-8 encoding, add the `{attribute 'monitoring_encoding' := 'UTF-8'}` pragma, page 618 to the declaration of a variable. It allows you to monitor the contents of the variable in UTF-8 encoding.

A STRING data type variable can contain any string of characters. The size entry in the declaration determines the memory space to be reserved for the variable. It refers to the number of bytes in the string and can be placed in parentheses or square brackets. If no size specification is given, the default size of 80 bytes will be used. If a variable is initialized with a string too long for the variable data type, the string will be correspondingly cut from right to left.

String length:

- For Latin 1 encoding, the memory space needed for a variable of type STRING is 1 byte per character + 1 additional byte. This means, the "`STRING[80]`" declaration needs 81 bytes.
- With UTF-8 encoding, one character can have a length of up to four bytes. The length of the string indicates the number of bytes which are reserved for the STRING. Thus, there is no longer a 1:1 relationship between the character count and the string length in bytes.

Example of a string declaration with 35 bytes:

```
str:STRING(35):='This is a String';
```

In general, the length of a string is not limited. When you execute string manipulations with the functions of the Standard library, strings up to a length of 255 are allowed. When you execute string manipulations with the functions of the StringUtils library, longer strings (>255 characters) are also permitted.

When a variable of the data type STRING is reinitialized by resetting the application, the contents of the old string, that exist after the terminating null character of the initial value, are not overwritten. This applies both to initialization with the initialization value and to initialization with the default initialization value of 0.

For further information, refer to WSTRING and STRING Constants (Operands), page 751.

> **NOTE:** You can use implicit checks to validate the conversion of variable types (refer to the chapter *POUs for Implicit Checks*, page 191).

WSTRING

The WSTRING data type differs from the STRING type (ASCII) by interpretation in Unicode format, and needing two bytes for each character and two bytes extra memory space for string termination (each only one in case of a STRING).

WSTRING data types are encoded as Unicode in UTF-16 format.

The library standard64.lib provides functions for WSTRING strings.

The number of characters for WSTRING depends on the contained characters. A size of 10 for WSTRING means that the length of the WSTRING can take a maximum of 10 WORDS. For some characters in Unicode, multiple WORDS are required for coding a character so that the number of characters does not have to correspond to the length of the WSTRING (10 in this case). The data type requires one WORD of extra memory as it is terminated with a `0`.

If a size is not defined, then 80 WORDS plus one for the terminating character `0` are allocated.

Examples:

```
wstr:WSTRING:="This is a WString";
wstr10 : WSTRING(10) := "1234567890";
```

For further information, refer to the following descriptions:

- STRING, page 638
- STRING constants, page 751 (operands)

Date and Time Data Types

The data types TIME, TIME_OF_DAY (shortened TOD), DATE, and DATE_AND_TIME (shortened DT) are handled internally like DWORD. The data types LDATE, LDATE_AND_TIME (LDT) and LTIME_OF_DAY (LTOD) are handled internally like LWORD.Time is given in milliseconds in TIME and TOD. Time in TOD begins at 12:00 A.M. Time is given in seconds in DATE and DT beginning with January 1, 1970 at 12:00 A.M.

LTIME is supported as time base for high resolution timers. LTIME is of size 64 bit and resolution nanoseconds.

Syntax of LTIME:

LTIME#<time declaration>

The time declaration can include the time units as used with the TIME constant and as:

- us : microseconds
- ns : nanoseconds

Example of LTIME:

```
LTIME1 := LTIME#1000d15h23m12s34ms2us44ns
```

For further information, refer to the following descriptions:

- Data Types, page 635
- TIME constants, page 746
- DATE constants, page 747
- DATE_AND_TIME constants, page 748
- TIME_OF_DAY constants, page 749
- LDATE constants, page 747
- LDATE_AND_TIME constants, page 748
- LTIME_OF_DAY constants, page 750

NOTE: You can use implicit checks to validate the conversion of variable types (refer to the chapter *POUs for Implicit Checks*, page 191).

ANY / ANY_<type>

When you implement a function, and one of the function inputs (*VAR_INPUT*) has a generic IEC data type (ANY or ANY_<type>), then the data type of the call parameter is not defined as unique. Variables of different data types can be passed to this function. The value passed and its type can be requested within the function with a predefined structure.

Generic IEC data types that allow the use of elementary data types, page 635 for function inputs:

Hierarchy of generic data types			Elementary data types
ANY	ANY_BIT	–	BYTE, WORD, DWORD, LWORD
	ANY_DATE	–	DATE_AND_TIME, DATE, TIME_OF_DAY, LDATE_ AND_TIME, LDATE, LTIME_OF_DAY,
	ANY_NUM	ANY_REAL	REAL, LREAL
		ANY_INT	USINT, UINT, UDINT, ULINT SINT, INT, DINT, LINT
	ANY_STRING	–	STRING, WSTRING

Example:

```
FUNCTION ANYBIT_TO_BCD : DWORD
   VAR_INPUT
      value : ANY_BIT;
   END_VAR
```

If the function ANYBIT_TO_BCD is called, then a variable of data type BYTE, WORD, DWORD, or LWORD can be passed to the function as a parameter.

Predefined structure:

When compiling the code, an ANY data type is replaced internally with the following structure:

```
TYPE AnyType :
STRUCT
        // the type of the actual parameter
    typeclass : __SYSTEM.TYPE_CLASS ;
        // the pointer to the actual parameter
    pvalue : POINTER TO BYTE;
        // the size of the data, to which the pointer points
    diSize : DINT;
END_STRUCT
END_TYPE
```

The actual call parameter assigns the structure elements at runtime.

Example:

This code example compares whether the two passed variables have the same type and the same value.

```
FUNCTION Generic_Compare : BOOL
VAR_INPUT
    any1 : ANY;
    any2 : ANY;
END_VAR
VAR
    icount: DINT;
END_VAR

Generic_Compare := FALSE;
IF any1.typeclass <> any2.typeclass THEN
    RETURN;
END_IF
IF any1.diSize <> any2.diSize THEN
    RETURN;
END_IF
```

```
// Byte comparison
FOR icount := 0 TO any1.diSize-1 DO
    IF any1.pvalue[iCount] <> any2.pvalue[iCount] THEN
        RETURN;
    END_IF
END_FOR
Generic_Compare := TRUE;
RETURN;
// END_FUNCTION
```

Also refer to the description of the __*VARINFO* operator, page 736.

UTF-8 Encoding

General Information

The STRING data type can be encoded in Latin 1 or in UTF-8 format. For details, refer to the paragraphs Project-Wide UTF-8 Encoding, page 643 and Encoding Single Literals in UTF-8 Format, page 644.

Since UTF-8 encoding provides the most comprehensive character set, it is a good practice to enable UTF-8 encoding for new projects, as well as for existing projects that are to be used in a new context.

EcoStruxure Machine Expert can process a wide variety of characters to display diagnostic messages or visualizations in different languages as well as to accept user input in these languages and characters or symbols.

If a comprehensive character set is not required or if a project is not subject to change, strings can be encoded in Latin 1 format. Additionally, the following character sets are supported:

Character set	Code page number	Description	Character encoding
ASCII	20127	• 128 characters • Suitable for English texts.	7-bit encoded character
DOS Latin 1	819, 850	• Complies with ISO/IEC 8859. • Suitable for Western European languages in the Windows command line.	8-bit encoded character
Latin 1	28591	• Complies with ISO/IEC 8859–1. • Frequently used for HTML pages with äöüß but without € or, for example, French special characters.	8-bit encoded character
Windows 1252 Encoding	1252	• Default Windows character set for Western European countries. • Windows internally uses the UTF-16 format. • Contains all characters from ISO/IEC 8859–1 and ISO/IEC 8859–15 but partly with different encoding.	8-bit encoded character
Unicode	-	• Universal character set for a wide range of languages, including historical languages, Braille, music, or emojis. • More than 100,000 characters can be displayed. • Each character has a numeric code. • In contrast to ASCII, a distinction is made between the assignment of code points to characters and the encoding of the characters. • Numeric codes < 128 are ASCII compatible • Numeric code < 256 are ISO/IEC 8859–1 compatible For more information, refer to https://home.unicode.org/.	-
Unicode 14.0	-	144,697 characters	-

Character set	Code page number	Description	Character encoding
UTF-16	1200	• Special Unicode • Used in some operating systems (Windows, OS X) and programming languages (Java, .NET) for internal character representation. • Different computer architectures encode the 4-byte characters differently. Little endian byte order for UTF-16LE.	16-bit encoded characters The characters are encoded either in 2 bytes or 4 bytes.
UTF-8	65001	• Byte-oriented encoding format of Unicode characters. • Most widely used. • Used in GNU/Linux and Unix operating systems, and in various Internet services (email, web, browser). • Compatible with ASCII characters in the first 128 characters (0...127).	Tuple of 8-bit words per character The characters are encoded in different lengths from 1...4 bytes.

Project-Wide UTF-8 Encoding

The **Project Settings > Compile options** dialog box provides the parameter **UTF8 Encoding for STRING** that allows you to configureUTF-8 encoding for all strings of data type STRING throughout the project. Refer to *Project Settings - Compile Options* in the Menu Commands Online Help.

> **NOTE:** Before you set the encoding format to UTF-8, execute the static analysis rule SA0175 (refer to the Menu Commands Online Help for details) on your code to help detect constructs that may cause issues with UTF-8 encoding.

WSTRING data types are not affected by this setting. They are always encoded as Unicode in UTF-16 format.

For data type STRING, project-wide encoding is as follows:

Option UTF8 Encoding for STRING	Project-wide encoding
Selected	UTF-8
Unselected	• Windows 1252 encoding (default Windows encoding) • Latin 1

If project-wide UTF-8 encoding is enabled, then this setting also applies to library functions and add-ons.

Encoding Single Literals in UTF-8 Format

If Latin 1 encoding is used throughout the project (**UTF8 Encoding for STRING** is unselected), you can encode single literals in UTF-8 format. To achieve this, add the `UTF8#` type prefix to each literal.

```
{attribute 'monitoring_encoding' := 'UTF-8'}
strVarUtf8: STRING := UTF8#'你好,世界!ÜüÄäÖö';
```

> **NOTE:** If you use single UTF-8 encoded strings, ensure that they are interpreted correctly wherever they are used.

Example: A string variable in the OPC UA server will be converted to UTF-8 before being transferred to a client if the setting is not selected. This leads to values such as `UTF8#'äöü'` being misinterpreted.

A similar condition can occur with strings that are displayed in the visualization.

For further information, refer to

- Constant: `UTF8#` String, page 753
- `Attribute monitoring_encoding`, page 618

String Manipulation

Use library functions to manipulate strings.

NOTICE
UNINTENDED STRING MANIPULATION Do NOT use an index access to a variable of type STRING for string manipulation as this may lead to unintended results when project-wide UTF-8 encoding is enabled. **Failure to follow these instructions can result in equipment damage.**

Extensions to IEC Standard

Overview

This chapter lists the data types that are supported in addition to the standard IEC 61131-3.

UNION

Overview

As an extension to the IEC 61131-3 standard, you can declare unions in user-defined types.

The components of a union have the same offset. Therefore, they occupy the same storage location. Thus, assuming a union definition as shown in the following example, an assignment to `utTest.udiValue` also manipulates `utTest.awValue`.

Example

```
TYPE UT_MyUnion: UNION
    udiValue : UDINT;
```

```
    awValue : ARRAY [0..1] OF WORD;
END_UNION
END_TYPE

VAR
    utTest: UT_MyUnion;
END_VAR
```

BIT

Overview

You can use the BIT data type only for particular variables within structures, page 661 or in a function block, page 192. The possible values are TRUE (1) and FALSE (0).

A BIT element consumes 1 bit of memory space and allows you to address single bits of a structure by name (for further information, refer to the paragraph *Bit Access in Structures*, page 663). Bit elements which are declared one after another will be combined in bytes. In contrast to BOOL types, page 635, where 8 bits are reserved in any case, the use of memory space can get optimized. On the other hand, the access to bits takes definitely more time. For that reason, use the BIT data type if you want to store several boolean pieces of information in a compact format.

References

Overview

This data type is available in extension to the IEC 61131-3 standard.

A reference stores the address of an object (a variable) that is located elsewhere in memory; in this respect, the behavior is identical to a pointer.

In contrast to a pointer, a variable behaves like an object when the syntax is concerned. Additionally, variables declared as REFERENCE provide the following advantages compared to POINTERS:

- A reference does not have to be dereferenced explicitly (with `^`) to access the contents of the referenced object.
- When passing values to input parameters of functions/function blocks/ methods, the following applies: If an input is declared as `REFERENCE TO <data type>`, a variable of the corresponding `<data type>` can be passed (`refInput :=` variable instead of `pInput :=ADR`(variable)).
- The compiler verifies that references of the same data type are assigned to each other.

For further information, refer to the *Assignment operator REF* description, page 363.

You can also use an implicit pointer monitoring function as described in the paragraph *CheckPointer* function, page 649.

Examples with `:=`

When references are assigned with `:=`, a value is always copied, regardless of whether the reference is on the left-hand side, on the right-hand side, or on both sides:

Reference at different positions	Copies the value	Pointer notation
`ref_int := iValue;`	The value `iValue` is written to the location to which the reference points to.	`p_int^ := iValue;`
`iValue := ref_int;`	The value to which the reference points to is written after `iValue`.	`iValue := p_int^;`
`ref_intA := ref_intB;`	The value to which `ref_intB` points to is written to the location to which `ref_intA` points to.	`p_intA^ := p_intB^;`

Examples with `REF=`

When references are assigned with `REF=`, the address is applied and a reference is required on the left-hand side:

Reference with `REF=`	Applies the address	Pointer notation
`ref_int REF= iValue;`	The reference points to `value`.	`p_int := ADR(iValue);`
`ref_intA REF= ref_intB;`	`ref_intA` points to the same value as `ref_intB`.	`p_intB := p_intB;`
`iValue REF= ref_int;`	A compile error is detected.	-

NOTE: In the declaration, `REFERENCE TO INT REF= iValue` behaves like `REFERENCE TO INT := iValue`.

Syntax

<identifier> : REFERENCE TO <data type>

Example Declaration

```
VAR
    p_int : POINTER TO INT;
    ref_int : REFERENCE TO INT;
    iVarA : INT;
    iVarB : INT;
END_VAR

ref_int REF= iVarA; // corresponds to p_int := ADR(iVarA)
ref_int := iVarB; // corresponds to p_int^ := iVarB)
```

NOTE: It is not possible to declare references like REFERENCE TO REFERENCE or ARRAY OF REFERENCE or POINTER TO REFERENCE.

Check for Valid References

You can use the operator __ISVALIDREF to check whether a reference points to a valid value that is a value unequal to 0.

Syntax

```
<Boolean variable name> := __ISVALIDREF( <reference name> );
```

<reference name>: Identifier declared with type REFERENCE TO <data type>.

The boolean variable is TRUE, if the reference points to a valid value. Otherwise it is FALSE.

Example

Declaration

```
ivar : INT;
ref_int : REFERENCE TO INT;
ref_int0: REFERENCE TO INT;
testref: BOOL := FALSE;
```

Implementation

```
ref_int REF= ivar;
ref_int0 REF= 0;
testref := __ISVALIDREF(ref_int);   (* will be TRUE, because
ref_int points to ivar, which is unequal 0 *)
testref := __ISVALIDREF(ref_int0); (* will be FALSE, because
ref_int is set to 0 *)
```

Pointers

Overview

As an extension to the IEC 61131-3 standard, you can use pointers.

Pointers save addresses while an application program is running. A pointer can point to a variable of any data type, page 635 or to a function block instance. The possibility of using an implicit pointer monitoring function is described further below in the paragraph *CheckPointer function*, page 649.

Syntax of a Pointer Declaration

```
<identifier>: POINTER TO <data type>;
```

Dereferencing a pointer means to obtain the value currently stored at the address to which it is pointing. You can dereference a pointer by adding the content operator ^ (ASCII caret or circumflex symbol), page 704 after the pointer identifier. See `pt^` in the example below.

You can use the `ADR` address operator, page 703 to assign the address of a variable to a pointer.

> **NOTE:** Pointers pointing to an I/O input require write access. This leads to the compiler message `'<pointer name>' is not a valid assignment target` when the code is generated.
>
> Example: `pwInput := ADR(wInput);`
>
> To help avoid the compiler detecting such an error, copy the input value (*wInput*) to a variable with write access.

Example

```
VAR
    piVar : POINTER TO INT;
    iVarA : INT;
    iVarB : INT;
END_VAR

iVarA := 5;
piVar := ADR(iVarA); // address of iVarA is stored in piVar
iVarB:= piVar^; // value of iVarA is stored in iVarB
```

In online mode, you can jump from a pointer to the declaration location of the referenced variable by executing the **Go To Reference** command (see Menu Commands, Online Help).

Function Pointers

Function pointers are also supported. These pointers can be passed to external libraries, but it is not possible to call a function pointer within an application in the programming system. The runtime function for registration of callback functions (system library function) expects the function pointer, and, depending on the callback for which the registration was requested, the respective function will be called implicitly by the runtime system (for example, at STOP). In order to enable such a system call (runtime system), set the respective properties (by default under **View > Properties... > Build**) for the function object.

You can use the `ADR` operator, page 703 on function names, program names, function block names, and method names. Since functions can move after online change, the result is not the address of the function, but the address of a pointer to the function. This address is valid as long as the function exists on the target.

Executing the **Online Change** command can change the contents of addresses.

⚠CAUTION

INVALID POINTER

Verify the validity of the pointers when using pointers on addresses and executing the Online Change command.

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

Index Access to Pointers

As an extension to the IEC 61131-3 standard, index access `[]` to variables of type POINTER, STRING, page 638 and WSTRING, page 639 is allowed.

- `[i]` returns the base data type.
- Index access on pointers is arithmetic:

  If the index access is used on a variable of type `POINTER OF <base type>`, the offset `piVar[1]` equates to `(piVar + i * SIZEOF(base type))^`. The index access also performs an implicit dereferencing on the pointer. The result type is the base type of the pointer.

  Note that `piVar[7] does not equate to (piVar + 7)^`.
- If the index access is used on a variable of type STRING, the result is of type BYTE.
- If the index access is used on a variable of type WSTRING, the result is of type WORD.

  NOTE: You can also use References, page 646. In contrast to a pointer, references directly affect a value.

Subtracting Pointers

The result of the difference between two pointers is a value of type DWORD. This also applies to 64-bit platforms when the pointers are 64-bit pointers.

`CheckPointer` Function

To monitor pointer access during runtime, you can use the implicit monitoring function *CheckPointer*. You can adapt it, if required. To achieve this, add the **POUs for implicit checks** object, page 191 to the application. Activate the check box related to the category **Pointer Checks**.

> **NOTE:** You have to implement the *CheckPointer* function during machine commissioning to verify whether the passed pointer refers to a valid memory address and whether the alignment of the referenced memory area fits to the data type of the variable that the pointer points to.

CheckPointer monitors variables of type REFERENCE TO in a similar way.

> **NOTE:** There is no implicit call of the check function for the `THIS` pointer.

Template:

Declaration part:

```
// Implicitly generated code : DO NOT EDIT
FUNCTION CheckPointer : POINTER TO BYTE
VAR_INPUT
ptToTest : POINTER TO BYTE;
iSize : DINT;
iGran : DINT;
bWrite: BOOL;
END_VAR
```

When called, the following input parameters are provided to the function:

- `ptToTest`: Target address of the pointer
- `iSize`: Size of referenced variable; the data type of `iSize` has to be integer-compatible and has to cover the maximum potential data size stored at the pointer address.
- `iGran`: Granularity of the access that is the largest non-structured data type used in the referenced variable; the data type of `iGran` has to be integer-compatible
- `bWrite`: Type of access (TRUE= write access, FALSE= read access); the data type of `bWrite` has to be BOOL.

Implementation of the `CheckPointer` Function for PacDrive Controllers

The implementation of the *CheckPointer* function for PacDrive controllers (PacDrive LMC Eco / PacDrive LMC Pro/Pro2) is to generate a system runtime exception and to write a call stack to the message logger when the memory address or the alignment is invalid.

Implementation part:

```
CheckPointer := ptToTest;
IF ptToTest = 0 THEN
        SystemInterface.FC_DiagMsgWrite(4, 'CP = 0');
        SystemInterface.FC_SysUserCallStack(0);
ELSE
        CheckPointer := ptToTest;
END_IF
```

Implementation of the `CheckPointer` Function for Optimized Controllers

The implementation of the *CheckPointer* function for Optimized Controllers (for example, Modicon M241 Logic Controller) is to return the passed pointer.

Implementation part (incomplete):

```
// No standard way of implementation. Fill your own code
here
CheckPointer := ptToTest;
```

In case of a positive result of the check, the unmodified input pointer will be returned (`ptToTest`).

User-Defined Data Types

Defined Data Types

Overview

Additionally, to the standard data types, you can define special data types within a project.

You can define them via creating DUT (Data Unit Type) objects, page 204 in the **POUs tree** or **Devices tree** or within the declaration part of a POU.

See the Best practices for the Naming of Identifiers, page 553 in order to make it as unique as possible.

See the following user-defined data types:

- arrays, page 651
- vector, page 654
- structures, page 661
- enumerations, page 663
- subrange types, page 666
- alias, page 668
- references, page 646
- pointers, page 647
- VERSION, page 670

Arrays

Overview

An array is a collection of data elements of the same data type. One- and multi-dimensional arrays of fixed or variable length are supported. You can define arrays both in the declaration part of a POU and in the global variable lists. You can also use implicit boundary checks, page 653.

The data type ARRAY with variable length can only be declared for *VAR_IN_OUT* variables of function blocks, methods, and functions. Use the operators `LOWER_BOUND(<array name>,<dim>)` and `UPPER_BOUND(<array name>,<dim>)` to get the lower and upper limits of this array.

NOTE: For arrays with variable length, statically declared arrays can be passed. Arrays generated by means of the operator `__NEW` are not allowed.

Syntax for the Declaration of an Array with Defined Length

<Array_Name> : ARRAY [<ll1>..<ul1>,<ll2>..<ul2>,..,<lln>..<uln>] OF <elem. type>

`ll1`, `ll2`, `lln` identify the lower limit of the field range.

`ul1`, `ul2`, and `uln` identify the upper limit of the field range.

The range values have to be of type integer.

Example for the Declaration of an Array with Defined Length

```
Card_game: ARRAY [1..13, 1..4] OF INT;
```

Syntax for the Declaration of an Array with Variable Length

```
<Array name> :ARRAY[* ( , * )+ ] OF <data type>;
```

`// (...)+`: One or more further dimensions

Example for the Declaration of an Array with Variable Length

```
FUNCTION SUM: INT; // Onedimensional
arrays of variable lengths can be
passed to this addition function.
VAR_IN_OUT
A: ARRAY [*] OF INT;
END_VAR
VAR
  i, sum2 : DINT;
END_VAR
sum2:= 0;
FOR i:= LOWER_BOUND(A,1) TO UPPER_BOUND(A, 1) DO // The
length of the respective array is determined.
  sum2:= sum2 + A[i];
END_FOR;
SUM:= sum2;
```

Initializing Arrays

Example for complete initialization of an array

```
arr1 : ARRAY [1..5] OF INT := [1,2,3,4,5];
arr2 : ARRAY [1..2,3..4] OF INT := [1,3(7)]; (* short for
1,7,7,7 *)
arr3 : ARRAY [1..2,2..3,3..4] OF INT := [2(0),4(4),2,3];
      (* short for 0,0,4,4,4,4,2,3 *)
```

Example of the initialization of an array of a structure

Structure definition

```
TYPE STRUCT1
STRUCT
 p1:int;
 p2:int;
 p3:dword;
END_STRUCT
END_TYPE
```

Array initialization

```
ARRAY[1..3] OF STRUCT1:= [(p1:=1,p2:=10,p3:=4723),(p1:=2,p2:
=0,p3:=299),(p1:=14,p2:=5,p3:=112)];
```

Example of the partial initialization of an array

```
arr1 : ARRAY [1..10] OF INT := [1,2];
```

Elements where no value is pre-assigned are initialized with the default initial value of the basic type. In the previous example, the elements `arr1[3]...arr1[10]` are therefore initialized with 0.

Example of the Initialization of an Array of Function Blocks with Additional Parameters in *FB_Init*

Example of a function block *FB* and method *FB_Init* with two parameters:

```
FUNCTION_BLOCK FB
VAR
    _nId : INT;
    _lrIn : LREAL;
END_VAR
METHOD FB_Init : BOOL
VAR_INPUT
    bInitRetains : BOOL;
    bInCopyCode : BOOL;
    nId : INT;
    lrIn : LREAL;
END_VAR
_nId := nId;
_lrIn := lrIN;
```

Example of array declaration with initialization:

```
PROGRAM PLC_PRG
VAR
  inst : FB(nId := 11, lrIn := 33.44);
    ainst : ARRAY [0..1, 0..1] OF FB[(nId :=
12, lrIn := 11.22), (nId := 13, lrIn :=
22.33), (nId := 14, lrIn := 33.55),(nId := 15,
lrIn := 11.22)];
END_VAR
```

Accessing Array Elements

In a two-dimensional array, access the elements as follows:

<Array name>[Index1,Index2]

Example:

```
Card_game [9,2]
```

Check Functions on Array Bounds

In order to access an array element properly during runtime, the function `CheckBounds` has to be available to the application. For information on inserting the function, refer to the description of the **POUs for implicit checks** function, page 191.

This check function has to treat boundary violations by an appropriate method (for example, by setting a detected error flag or adjusting the index). The function is called implicitly as soon as a variable of type ARRAY is assigned.

Example for the Use of Function *CheckBounds*

The default implementation of the check function is the following:

Declaration part:

```
// Implicitly generated code : DO NOT EDIT
FUNCTION CheckBounds : DINT
VAR_INPUT
index, lower, upper: DINT;
END_VAR
```

Implementation part:

```
// Implicitly generated code : Only an Implementation
suggestion
IF  index < lower THEN
CheckBounds := lower;
ELSIF  index > upper THEN
CheckBounds := upper;
ELSE
CheckBounds := index;
END_IF
```

When called, the function gets the following input parameters:

- *index*: field element index
- *lower*: the lower limit of the field range
- *upper*: the upper limit of the field range

As long as the index is within the range, the return value is the index itself. Otherwise, in correspondence to the range violation either the upper or the lower limit of the field range will be returned.

Exceeding the Upper Limit of the Array *a*

The upper limit of the array *a* is exceeded in the following example:

```
PROGRAM PLC_PRG
VAR
 a: ARRAY[0..7] OF BOOL;
    b: INT:=10;
END_VAR
a[b]:=TRUE;
```

In this example, the implicit call to the `CheckBounds` function preceding the assignment affects that the value of the index is changed from `10` into the upper limit `7`. Therefore, the value TRUE is assigned to the element *a[7]* of the array. This is how you can correct attempted access outside the array range via the function `CheckBounds`.

Vector

Overview

Vector operations are supported natively only on 64-bit processors and offer a performance advantage only on these processors. Consult the data sheet of your controller for information on the processor.

Vector operations on the x86/64-bit platforms with SSE2 and ARM64 with NEON are supported natively. On other platforms, vector operations are divided into individual statements. For example, vector addition is performed by executing multiple single addition operations.

The command set extensions of the processors are SIMD (Single Instruction, Multiple Data) extensions. SIMD describes a computer architecture in which multiple data sets of the same type are processed simultaneously with one command call. In vector operations, for example, it is possible to add four pairs of numbers at the same time.

Syntax

```
<variable name> :  __VECTOR[ <vector size> ] OF <element
type> ( := <initialization> )? ;

<vector size> : 1 |2 | 3 | 4 | 5| 6 | 7| 8
<element type> : REAL | LREAL
// (...)? : Optional
```

A vector data type is an array of floating-point numbers with a maximum of 8 elements. The operators `__vc<operator name>` are available for this data type. They allow you to implement vector operations without additional function calls.

Syntax for Index Access

```
<variable name>[ <index> ]
<index> : 0 | 1 | 2| 3 | 4 | 5| 6 | 7
```

When indexing a vector variable, you can access a single element of the vector. The index range is from 0...<vector size> - 1.

Example:

```
PROGRAM PLC_PRG
VAR
        vcA : __VECTOR[3] OF REAL;
END_VAR

vcA[0] := 1.1;
vcA[1] := 2.2;
vcA[2] := 3.3;
```

Determining the Optimal Vector Size for Your System

For programming efficient code, determine and use the vector size that suits your target system.

For target systems with computer architecture that is generally suitable for vector processing, it is not a good practice to use vectors of arbitrary size. An optimal vector size is available for each processor depending on the type of data processing. The processing time of vectors of this size is optimized. There is no advantage in speed for vectors that are declared as a larger array. Vectors that are declared as smaller arrays do not fully exploit the capabilities of the processor.

You can query the optimal size at runtime by using the following constants of data type INT:

- `Constants.vcOptimalREAL`: For vectors with REAL elements.
- `Constants.vcOptimalLREAL`: For vectors with LREAL elements.

A return value `1` means that the accelerated vector processing is not available for the target system.

Example:

```
PROGRAM PLC_PRG
VAR
        iOVS_REAL : INT; // Optimal vector size for REAL
elements
        iOVS_LREAL : INT; // Optimal vector size for LREAL
elements
END_VAR

iOVS_REAL := Constants.vcOptimalREAL;
iOVS_LREAL := Constants.vcOptimalLREAL;
```

Operator __*VCADD*

The operator calculates the sum of two vectors.

Syntax:
```
<vector variable> := <1st vector operand> __VCADD <2nd
vector operand>;
```

Example of an add operation:
```
FUNCTION_BLOCK FB_ADD
VAR
        vcA : __VECTOR[3] OF REAL := __VCSET_REAL(3, 3, 3);
        vcB : __VECTOR[3] OF REAL := __VCSET_REAL(1, 2, 3);
        vcResult : __VECTOR[3] OF REAL;
END_VAR


vcResult := vcA __VCADD vcB;
```

Operator __*VCSUB*

The operator calculates the difference between two vectors.

Syntax:
```
<vector variable> := <vector minuend> __VCSUB <vector
subtrahend>;
```

Example of a subtract operation:
```
FUNCTION_BLOCK FB_SUB
VAR
        vcA : __VECTOR[3] OF REAL := __VCSET_REAL(3, 3, 3);
        vcB : __VECTOR[3] OF REAL := __VCSET_REAL(1, 2, 3);
        vcResult0 : __VECTOR[3] OF REAL;
        vcResult1 : __VECTOR[3] OF REAL;
END_VAR

vcResult0 := vcA __VCSUB vcB;
vcResult1 := vcB __VCSUB vcA;
```

Operator __*VCMUL*

The operator calculates the product of two vectors or of a scalar (floating-point number) and a vector.

Syntax:
```
<vector variable> := <1st vector operand> __VCMUL <2nd
vector operand> | <scalar operand> __VCMUL <vector operand>
| <vector operand> __VCMUL <scalar operand>;
```

Example of a multiplication operation:
```
FUNCTION_BLOCK FB_MUL
VAR
        rScalar : REAL := 1.1;
        vcA : __VECTOR[3] OF REAL;
        vcB : __VECTOR[3] OF REAL;
        vcResult0 : __VECTOR[3] OF REAL;
        vcResult1 : __VECTOR[3] OF REAL;
        vcResult2 : __VECTOR[3] OF REAL;
END_VAR

vcResult0 := vcA __VCMUL vcB;
vcResult1 := rScalar __VCMUL vcB;
vcResult2 := vcA __VCMUL 3.3;
```

Operator __VCDIV

The operator calculates the quotient of two vectors or of a vector and a scalar.

Syntax:
```
<vector variable> := <vector dividend> __VCDIV <vector
divisor> | <vector dividend> __VCMUL <scalar divisor>;
```

Example of a division operation:
```
FUNCTION_BLOCK FB_DIV
VAR
        iScalar : INT := 3;
        rScalar : REAL := 1.5;
        vcA : __VECTOR[3] OF REAL := __VCSET_REAL(3, 3, 3);
        vcB : __VECTOR[3] OF REAL := __VCSET_REAL(1, 2, 3);
        vcResult0 : __VECTOR[3] OF REAL;
        vcResult1 : __VECTOR[3] OF REAL;
        vcResult2 : __VECTOR[3] OF REAL;
END_VAR

vcResult0 := vcA __VCDIV vcB;
// ERROR CODE vcResult1 := rScalar __VCDIV vcB;
// ERROR CODE vcResult1 := iScalar __VCDIV vcB;
// ERROR CODE vcResult1 := 3.3 __VCDIV vcB;
vcResult2 := vcA __VCDIV 1.5;
vcResult2 := vcA __VCDIV iScalar;
vcResult2 := vcA __VCDIV rScalar;
```

Operator __VCDOT

The operator calculates the dot product (scalar product) of two vectors.

Syntax:
```
<scalar variable> := <1st vector operand> __VCDOT <2nd
vector operand>;
```

Example of a dot product:
```
FUNCTION_BLOCK FB_DOT
VAR
        rResult : REAL;
        vcA : __VECTOR[3] OF REAL := __VCSET_REAL(3, 3, 3);
        vcB : __VECTOR[3] OF REAL := __VCSET_REAL(1, 2, 3);
END_VAR

rResult := vcA __VCDOT vcB; // = 18
```

Operator __VCSQRT

The operator calculates the square root of each element in the vector.

Syntax:
```
<vector variable> := __VCSQRT <vector operand>;
```

Example of a square root:
```
FUNCTION_BLOCK FB_SQRT
VAR
        vcA : __VECTOR[3] OF REAL := __VCSET_REAL(4, 9, 16);
        vcResult0 : __VECTOR[3] OF REAL;
END_VAR

vcResult0 := __VCSQRT(vcA);
```

Operator __*VCMAX*

The operator calculates the maximum vector of two vectors. The maximum is determined element by element.

Syntax:
```
<vector variable> := __VCMAX <1st vector operand>, <2nd
vector operand>;
```

Example of a maximum vector:
```
FUNCTION_BLOCK FB_MAX
VAR
        vcA : __VECTOR[3] OF REAL := __VCSET_REAL(3, 3, 3);
        vcB : __VECTOR[3] OF REAL := __VCSET_REAL(1, 2, 6);
        vcResult0 : __VECTOR[3] OF REAL;
END_VAR

vcResult0 := __VCMAX(vcA, vcB);
```

Operator __*VCMIN*

The operator calculates the minimum vector of two vectors. The minimum is determined element by element.

Syntax:
```
<vector variable> := __VCMIN <1st vector operand>, <2nd
vector operand>;
```

Example of a minimum vector:
```
FUNCTION_BLOCK FB_MIN
VAR
        vcA : __VECTOR[3] OF REAL := __VCSET_REAL(3, 3, 3);
        vcB : __VECTOR[3] OF REAL := __VCSET_REAL(1, 2, 6);
        vcResult0 : __VECTOR[3] OF REAL;
END_VAR

vcResult0 := __VCMIN(vcA, vcB);
```

Operator __*VCSET_REAL*

The operator sets the elements of a vector in a statement. The elements are of data type REAL.

Syntax:
```
<vector variable> __VCSET_REAL( <first literal>, ( < next
literal> )+ ) ;
( ... )+ // number of elements have to match
```

Example:
```
FUNCTION_BLOCK FB_SET
VAR
        vcA : __VECTOR[3] OF REAL := __VCSET_REAL(3, 3, 3);
        vcB : __VECTOR[3] OF REAL := __VCSET_REAL(1, 2, 3);
END_VAR

vcA := __VCSET_REAL(4, 4, 4);
vcB := __VCSET_REAL(1.1, 2.2, 3.3);
```

Operator __VCSET_LREAL

The operator sets the elements of a vector in a statement. The elements are of data type LREAL.

You can use the operator wherever variables are valid, such as in assignments, implementations or as parameters in function calls.

Syntax:
```
<vector variable> __VCSET_LREAL( <first literal>, ( < next
literal> )+ ) ;
( ... )+ // number of elements have to match
```

Example:
```
FUNCTION_BLOCK FB_SET
VAR
        vclA : __VECTOR[3] OF LREAL := __VCSET_LREAL(3, 3,
3);
        vclB : __VECTOR[3] OF LREAL := __VCSET_LREAL(1, 2,
3);
END_VAR

vclA := __VCSET_LREAL(-1.7976931348623158E+308, 0.0,
1.7976931348623158E+308);
vclB := __VCSET_LREAL(-1.7976931348623158E+308, 0.0,
1.7976931348623158E+308);
```

Operator __VCLOAD_REAL

The operator interprets each arbitrary memory area as a vector. This is useful for connecting vector variables to existing code.

The operator requires two parameters:

- The first parameter indicates the number of vector elements.
- The second parameter is a pointer to the REAL data.

__VCLOAD_REAL (<n>,<ptr>) reads <n> successive REAL values at the <ptr> address. Ensure that there are <n> values at this address.

⚠CAUTION

INVALID POINTER

Verify the validity of the pointers when using pointers on addresses and executing the Online Change command.

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

Syntax:
```
<vector variable> __VCLOAD_REAL( <vector size>, ( < pointer
to data of type REAL> ) ;
<vector size> : 2 | 3 | 4 | 5| 6 | 7| 8
```

Example of interpretation as vector:
```
FUNCTION_BLOCK FB_LOAD
VAR_INPUT
END_VAR
VAR_OUTPUT
END_VAR
VAR
        rData : ARRAY[0..2] OF REAL := [1.234, 5.678,
9.123];
        vcA : __VECTOR[3] OF REAL := __VCSET_REAL(3, 3, 3);
END_VAR

vcA := __VCLOAD_REAL(3, ADR(rData[0]));
```

Operator __*VCLOAD_LREAL*

The operator interprets each arbitrary memory area as a vector. This is useful for connecting vector variables to existing code.

The operator requires two parameters:

- The first parameter indicates the number of vector elements.
- The second parameter is a pointer to the LREAL data.

__VCLOAD_LREAL (<n>,<ptr>) reads <n> successive LREAL values at the <ptr> address. Ensure that there are <n> values at this address.

⚠CAUTION

INVALID POINTER

Verify the validity of the pointers when using pointers on addresses and executing the Online Change command.

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

Syntax:

```
<vector variable> __VCLOAD_LREAL( <vector size>, ( < pointer
to data of type LREAL> ) ;
<number of vector elements> : 1 | 2 | 3 | 4 | 5| 6 | 7| 8
```

Example of interpretation as vector:

```
FUNCTION_BLOCK FB_LOAD
VAR_INPUT
END_VAR
VAR_OUTPUT
END_VAR
VAR
        rData : ARRAY[0..3] OF LREAL :=
[-1.7976931348623158E+308, 1.6E+308, 1.7E+308, -1.6E+308];
        vcA : __VECTOR[3] OF LREAL := __VCSET_LREAL(1, 2,
3);
END_VAR
vcA := __VCLOAD_LREAL(3, ADR(rData[0]));
```

Operator __*VCSTORE*

The operator saves/copies the contents of the vector to the specified memory address. The number and the types of elements from the vector variables are applied.

__VCSTORE(<n>,<ptr>) writes <n> successive values at the <ptr> address. Ensure that there are <n> values at this address.

⚠CAUTION

INVALID POINTER

Verify the validity of the pointers when using pointers on addresses and executing the Online Change command.

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

Syntax:

```
__VCSTORE( <pointer to data>, <vector variable> );
```

Example of storage:

```
FUNCTION_BLOCK FB_STORE
VAR_INPUT
END_VAR
VAR_OUTPUT
END_VAR
VAR
        rData : ARRAY[0..2] OF REAL := [1, 2, 3];
        pData: POINTER TO REAL := ADR(rData[0]);

        lrData : ARRAY [0..3] OF LREAL := [1, 2, 3, 4];
        plData: POINTER TO LREAL := ADR(lrData[0];


        vcA : __VECTOR[3] OF REAL := __VCSET_REAL( 1.234,
5.678, 9.123);
        vclA : __VECTOR[4] OF LREAL := __VCSET_LREAL
(-1.7976931348623158E+308, 1.6E+308, 1.7E+308, -1.6E+308);
END_VAR

__VCSTORE(pData, vcA);
__VCSTORE(plData, vclA);
```

Structures

Overview

Create structures in a project as DUT (Data Type Unit) objects via the **Add Object** dialog box.

They begin with the keywords `TYPE` and `STRUCT` and end with `END_STRUCT` and `END_TYPE`.

Syntax

```
TYPE <structurename>:
STRUCT
    <declaration of variables 1>
    ...
    <declaration of variables n>
END_STRUCT
END_TYPE
```

<structurename> is a type that is recognized throughout the project and can be used like a standard data type.

Nested structures are allowed. The only restriction is that variables may not be assigned to addresses (the `AT` declaration is not allowed).

Example

Example for a structure definition named `Polygonline`:

```
TYPE S_POLYGONLINE :
STRUCT
        aiStart : ARRAY[1..2] OF INT := [-99, -99];
        aiPoint1 : ARRAY[1..2] OF INT;
        aiPoint2 : ARRAY[1..2] OF INT;
        aiPoint3 : ARRAY[1..2] OF INT;
        aiPoint4 : ARRAY[1..2] OF INT;
        aiEnd : ARRAY[1..2] OF INT := [99, 99];
END_STRUCT
END_TYPE
```

Initialization of Structures

Example:

```
Poly_1:polygonline := ( Start:=[3,3], Point1:=[5,2], Point2:
=[7,3], Point3:=[8,5], Point4:=[5,7], End:= [3,5]);
```

Initializations with variables are not possible. For an example of the initialization of an array of a structure, refer to *Arrays*, page 651.

Access on Structure Components

You can gain access to structure components using the following syntax:

<structurename>.<componentname>

For the previous example of the structure `Polygonline`, you can access the component `Start` by `Poly_1.Start`.

Bit Access in Structures

The data type BIT, page 645 is a special data type which can only be defined in structures. It consumes memory space of 1 bit and allows you to address single bits of a structure by name.

```
TYPE <structurename>:
STRUCT
    <bitname bit1> : BIT;
    <bitname bit2> : BIT;
    <bitname bit3> : BIT;
    ...
    <bitname bitn> : BIT;
END_STRUCT
END_TYPE
```

You can gain access to the structure component `BIT` by using the following syntax:

<structurename>.<bitname>

NOTE: The usage of references and pointer on `BIT` variables is not possible. Furthermore, `BIT` variables are not allowed in arrays.

Enumerations

Overview

An enumeration is a user-defined type that is made up of a number of comma-delimited string constants. These constants are referred to as enumeration values. Enumeration values are identifiers for global constants in the project.

Enumeration values are recognized globally in all areas of the project even if they are declared within a POU.

An enumeration is created in a project as a DUT object via the **Add Object** dialog box.

NOTE: Local enumeration declaration is only possible within TYPE.

Syntax for Declaring Enumerations

```
TYPE <enum identifier>: (<enum_0>|:=<value>,<enum_1>|:
=<value>, ...,<enum _n>|:=<value>)|<base data type>|:
=<default value>;
END_TYPE
```

The following elements can additionally be used:

- Initialization of single enumeration values
- Definition of a `<base data type>` (refer to the paragraph *Second Extension to the IEC 61131-3 Standard*, page 665)
- Definitions of a default value for initializing all components

Syntax for Declaring Variables with the Enumeration Type

```
<variable identifier> : <enum identifier>| :=
<initialization value>
```

A variable of type `<enum identifier>` can take the enumeration values `<enum_..>`.

Initialization

If...	Then ...
• If you have not defined a `<default value>` in the declaration of the enumeration (see following example), and • If you have neither defined an explicit initialization value for the declaration of variables of the type `<enum identifier>`	The variable is initialized with the value of the first enumeration component. This applies unless a component is initialized with the value 0 in the enumeration declaration. In this case, the variable is also initialized with the value 0.

Example of an Enumeration with an Explicit Initialization Value for a Component

Declaration of the Enumeration

```
TYPE TRAFFIC_SIGNAL: (red, yellow, green:=10);
    (* The initial value for each of the colors is red 0,
yellow 1, green 10 *)
END_TYPE
```

In this declaration, the first two components get default initialization values:

`red` = 0, `yellow` = 1

The initialization value of the third component is explicitly defined:

`green` = 10

Use in the Implementation

```
TRAFFIC_SIGNAL1 : TRAFFIC_SIGNAL;
TRAFFIC_SIGNAL1:=10; (* The value of the TRAFFIC_SIGNAL1 is
"green" *)
FOR i:= red TO green DO
    i := i + 1;
END_FOR;
```

Example of an Enumeration with a Defined Default Value

Declaration of the Enumeration

```
TYPE COLOR :
(
    red,
    yellow,
    green
) := green;
END_TYPE
```

Use in the Implementation

```
c1 : COLOR;
c2 : COLOR := yellow;
```

In this case, the variable `c1` is initialized with the value `green`. The initialization value `yellow` is defined for `c2`.

First Extension to the IEC 61131-3 Standard

You can use the type name of enumerations (as a scope operator, page 741) to disambiguate the access to an enumeration constant.

Therefore, it is possible to use the same constant in different enumerations.

Example

Definition of two enumerations with same-named components

```
TYPE COLORS_1: (red, blue);
END_TYPE
```

```
TYPE COLORS_2: (green, blue, yellow);
END_TYPE
```

Use of same-named components from different enumerations in one block

```
colorvar1 : COLORS_1;
colorvar2 : COLORS_2;
(* valid: *)
colorvar1 := COLORS_1.blue;
colorvar2 := COLORS_2.blue;
(* invalid: *)
colorvar1 := blue;
colorvar2 := blue;
```

Second Extension to the IEC 61131-3 Standard

You can specify explicitly the base data type of the enumeration, which by default is INT.

Example of an Explicit Other Base Data Type for an Enumeration

```
TYPE COLORS_2 : (yellow, blue, green:=16#8000)DINT;
END_TYPE
```

NOTE: The `strict` attribute is automatically assigned to each enumeration that you add to a project in the line above the `TYPE` declaration. This leads to errors that are detected during the compile process in the following cases:

- Arithmetic operation with variables of the enumeration type
- Assignment of a constant value to a variable of the enumeration type, in which the constant does not correspond to an enumeration value
- Assignment of a non-constant value to a variable of the enumeration type, in which the non-constant has another data type than the enumeration type

You can explicitly add or remove the attribute.

Syntax: `{attribute 'strict'}`

Implicit Enumerations

Overview

The implicit enumeration declares an enumeration variable that is valid within the local POU. A variable of this type can take a limited number of integer values, which are declared in the component list.

The implicit enumeration does not have a data type name or identifier.

Declaration

```
<variable name> : ( <list of component names> ) :=
<initialization>;
```

`<list of component names>`	Comma-delimited list of component names containing a number of components of your choice Each component represents an integer value, starting at 0. The value of each following component is incremented by 1. Example: `Alfa, Bravo, Charlie, Delta, Echo` A fixed value can also be assigned to each component. Example: `Zero, Alfa:=10, Bravo:=20, Charlie:=30, Delta:=40, Echo`
`<initialization>`	An optional initialization value, usually a component

Call Example

```
PROGRAM PLC_PRG
VAR
    iAlphabet : (Alfa, Bravo, Charlie, Delta, Echo) := Echo;
    strText : STRING;
    iNumber : INT;
END_VAR
CASE iAlphabet OF
    Alfa:
    strText := 'A';

    Bravo:
    strText := 'B';

    Charlie:
    strText := 'C';

    Delta:
    strText := 'D';

    Echo:
    strText := 'E';
END_CASE
iNumber := Alphabet;
```

Internal Data Type

An implicit enumeration creates an internal data type assigning a name according to the following scheme:
`IMPLICIT_ENUM_ <POU name> _ <variable name>`

Do not use this internal identifier because it is possible that the internal identifier is different.

Subrange Types

Overview

A subrange type is a user-defined type, page 650 whose range of values is only a subset of that of the elementary data type, page 635. You can also use implicit range boundary checks, page 667.

You can do the declaration in a DUT object but you can also declare a variable directly with a subrange type.

Syntax

Syntax for the declaration as a DUT object:

TYPE <name>: <Inttype> (<ug>..<og>) END_TYPE;

`<name>`	a valid IEC identifier
`<inttype>`	one of the data types SINT, USINT, INT, UINT, DINT, UDINT, BYTE, WORD, DWORD, LINT, ULINT, LWORD
`<ug>`	a constant compatible with the elementary type, setting the lower boundary of the range types The lower boundary itself is included in this range.
`<og>`	a constant compatible with the elementary type, setting the upper boundary of the range types. The upper boundary itself is included in this elementary type.

Example

```
TYPE
 SubInt : INT (-4095..4095);
END_TYPE
```

Direct Declaration of a Variable with a Subrange Type

```
VAR
 i : INT (-4095..4095);
 ui : UINT (0..10000);
END_VAR
```

If a value is assigned to a subrange type (in the declaration or in the implementation) but does not match this range (for example, `i:=5000` in the upper shown declaration example), a message will be issued.

Check Functions for Range Bounds

In order to observe the range bounds of subrange types during runtime, the functions `CheckRangeSigned`, `CheckLRangeSigned` or `CheckRangeUnsigned`, `CheckLRangeUnsigned` have to be added to the application. For information on inserting the function, refer to the description of the **POUs for implicit checks** function, page 191.

The purpose of this check function is the proper treatment of violations of the subrange (for example, by setting an error flag or changing the value). The function is called implicitly as soon as a variable of subrange type is assigned.

⚠WARNING

UNINTENDED EQUIPMENT OPERATION

Do not change the declaration part of an implicit check function.

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

Example

The assignment of a variable belonging to a signed subrange type entails an implicit call to `CheckRangeSigned`. The default implementation of that function trimming a value to the permissible range is provided as follows:

Declaration part:

```
// Implicitly generated code : DO NOT EDIT
FUNCTION CheckRangeSigned : DINT
VAR_INPUT
value, lower, upper: DINT;
END_VAR
```

Implementation part:

```
// Implicitly generated code : Only an Implementation
suggestion
IF (value < lower) THEN
CheckRangeSigned := lower;
ELSIF(value > upper) THEN
CheckRangeSigned := upper;
ELSE
CheckRangeSigned := value;
END_IF
```

When called, the function gets the following input parameters:

- `value`: the value to be assigned to the range type
- `lower`: the lower boundary of the range
- `upper`: the upper boundary of the range

As long as the assigned value is within the valid range, it will be used as return value of the function. Otherwise, in correspondence to the range violation, either the upper or the lower boundary of the range will be returned.

The assignment `i:=10*y` will now be replaced implicitly by
`i := CheckRangeSigned(10*y, -4095, 4095);`

If `y`, for example, has the value `1000`, the variable `i` will not be assigned to 10*1000=10000 (as provided by the original implementation), but to the upper boundary of the range that is 4095.

The same applies to function `CheckRangeUnsigned`.

> **NOTE:** If neither of the functions is available, no type checking of subrange types occurs during runtime. In this case, you can assign any DINT/UDINT value to a variable of subrange type DINT/UDINT. You can assign any LINT/ULINT value to a variable of a subrange type LINT/ULINT.

Alias

Overview

A user-defined data type alias allows you to declare an alternative name for a basic type, data type, or function block.

Select the **Type** = **Alias** in the **Add DUT** dialog box, page 204.

Declaration of an Alias in a DUT

```
TYPE A_MESSAGE : STRING[50];
END_TYPE
```

Syntax

```
TYPE <DUT name> : <type name>;
END_TYPE
```

The following types are permitted for <type name>:

- <basic type name>
- <data type name>
- <function block name>

Example for Variables of Type STRING or ARRAY with Specific Length

Declaration part:

```
TYPE FRAME : ARRAY[0..1499] OF BYTE; END_TYPE
TYPE SYMBOL : STRING(512); END_TYPE
```

Call:

```
PROGRAM PLC_PRG
VAR
    frameF : FRAME;
    symbolS : SYMBOL;
END_VAR
```

Example for Variables Requiring a Different Initial Value

This example is valid for an alias data type for variables that require a different initial value than the one provided by the compiler.

```
TYPE INDEX : DINT := -1; END_TYPE
```

Example for Variables Applying a Subset of Values

This example is valid for an alias data type for variables of a specific type where a specific subset of values of the original type is to be applied.

```
{attribute 'qualified_only'}
VAR_GLOBAL CONSTANT
    c_diMaxRune : DINT := DINT#16#0010FFFF;
END_VAR
TYPE RUNE : DINT(0..GVL.c_diMaxRune); END_TYPE
```

Example for Types From Another Namespace

This example is valid, for instance, for types from a subordinate library to make them available in the present namespace.

```
TYPE ENCODING: SBB.ENCODING; END_TYPE (*ENUM*)
TYPE RUNE : UTF8.RUNE;  END_TYPE (*SUBTYPE*)
TYPE INFO : STR.INFO;  END_TYPE (*STRUCT*)
TYPE IBuilder : SBB.IBuilder; END_TYPE (*INTERFACE*)
TYPE Range: SBB.Range; END_TYPE (*FUNCTION_BLOCK*)
```

Since the types of subordinate libraries become part of your own library, a container library becomes superfluous in most cases. For further information, refer to the Functions and Libraries User Guide, chapter *Create Your Own Library, Step 3: Reference Other Libraries, if Needed*.

VERSION

Overview

A user-defined data type VERSION contains a structure providing version information about the project or library. If the **Automatically generate 'Project Information' POUs** option and/or **Automatically generate 'Library Information' POUs** option is activated in the **Summary** tab of the **Project Information** (refer to the Menu Commands Online Help for further information), the **GetVersion** function with the VERSION data type is automatically generated in the **Global** node of the **Tools tree**. The individual structural elements can be accessed programmatically.

Type Declaration

```
TYPE VERSION:
STRUCT
    /// The major version number. This is typically incremented for significant
releases that may include major new features or changes.
    uiMajor : UINT;
    /// The minor version number. This is incremented for releases that add
smaller features or improvements and may include significant fixes.
    uiMinor : UINT;
    /// The service pack number. This is used to indicate updates that usually
include bug fixes, performance improvements, and minor feature enhancements.
    uiServicePack : UINT;
    /// The patch number. This is incremented for very small changes or bug fixes
that do not significantly alter the functionality.
    uiPatch : UINT;
    END_STRUCT
    END_TYPE
```

Programming Guidelines

What's in This Chapter

Naming Conventions 671
Prefixes 672
SmartTag Functions 677

Naming Conventions

General Information

Creating Designator Names

Choose a relevant, short, description in English for each designator: the basis name. The basis name should be self-explanatory. Capitalize the first letter of each word in the basis name. Write the rest in lower case letters (example: `FileSize`). This basis name receives prefixes to indicate scope and properties.

Whenever possible, the designator should not contain more than 20 characters. This value is a guideline. You can adjust the number upward or downward if necessary.

If abbreviations of standard terms (TP, JK-FlipFlop, ...) are used, the name should not contain more than 3 capital letters in sequence.

Case-Sensitivity

Consider case-sensitivity, especially for prefixes, to improve readability when using designators in the IEC program.

NOTE: The compiler is not case-sensitive.

Valid Characters

Use only the following letters, numbers and special characters in designators:

0...9, A...Z, a...z,

In order to be able to display the prefixes clearly, an underline is used as the separator. The syntax is explained in the respective prefix section.

Do not use underscores in the basis name.

Examples

Correct Designator	Incorrect Designators
`diState`	`diSTATE`
`xInit`	`x_Init`
`diCycleCounter`	`diCyclecounter`
`lrRefVelocity`	`lrRef_Velocity`
`c_lrMaxPosition`	`clrMaxPosition`
`FC_PidController`	`FC_PIDController`

Prefixes

Prefix Parts

Overview

Prefixes are used to assign names by function.

The following parts of prefixes are available:

Prefix part	Use	Syntax	Example
scope prefix, page 673	scope of variables and constants	[scope]_[designator]	`G_diFirstUserFault`
data type prefix, page 674	identifying the data type of variables and constants	[type][designator]	`xEnable`
property prefix, page 675	identifying the properties of variables and constants	[property]_[designator]	`c_iNumberOfAxes`
POU prefix, page 676	identifying if POU was implemented as a function, function block, or program	[POU]_[designator]	`FB_VisuController`
namespace prefix, page 676	for POUs, data types, variables, and constants declared within a library	[namespace].[identifier]	`TPL.G_dwErrorCode`

Order of Prefixes

Overview

Designators contain the scope prefix and the type prefix. Use the property prefix according to the property of the variables (for example, for constants). An additional namespace prefix is used for libraries.

Obligatory Order

The following order is obligatory:

scope][property][_][type][identifier]

Scope prefixes and property prefixes are separated from type prefixes by an underscore (_).

Example

```
Gc_dwErrorCode    : DWORD;
diCycleCounter    : DINT;
```

The additional namespace prefix is used for libraries:

[namespace].[scope][property][_][type][identifier]

Example

```
ExampleLibrary.Gc_dwErrorCode
```

Independent Program Organization Units (POUs)

Insert an underscore to separate program organization units (functions, function blocks, and programs) prefixes from identifiers:

[POU][_][identifier]

Example

```
FB_MotionCorrection
```

Use the additional namespace prefix for libraries:

[namespace].[POU][_][identifier]

Namespace prefixes are separated from POU prefixes by a dot (`.`).

Example

```
ExampleLibrary.FC_SetError()
```

Dependent Program Organization Units (POUs)

Methods, actions, and properties are considered to be dependent POUs. These are used on a level below an independent POU.

Methods and actions do not have any prefixes.

Properties receive the type prefix of their return value.

Example

```
PROPERTY lrVelocity : LREAL
```

Scope Prefix

Overview

The scope prefix indicates the scope of variables and constants. It indicates whether it is a local or global variable, or a constant.

Global variables are indicated by a capital `G_` and a property prefix `c` is added to global constants (followed by an underscore in each case).

> **NOTE:** Additionally identify the global variables and constants of libraries with the namespace of the library.

Scope Prefix	Type	Use	Example
no prefix	`VAR`	local variable	`xEnable`
`G_`	`VAR_GLOBAL`	global variable	`G_diFirstUserFault`
`Gc_`	`VAR_GLOBAL CONSTANT`	global constant	`Gc_dwErrorCode`

Example

```
VAR_GLOBAL CONSTANT
    Gc_dwExample : DWORD := 16#0000001A;
END_VAR
```

Access to the global variable of a library with the namespace `INF`:

```
INF.G_dwExample := 16#0000001A;
```

Data Type Prefix

Standard Data Types

The data type prefix identifies the data type of variables and constants.

NOTE: The data type prefix can also be composite, for example, for pointers, references and arrays. The pointer or array is listed first, followed by the prefix of the pointer type or array type.

The IEC 61131-3 standard data type prefixes as well as the prefixes for the extensions to the standard are listed in the table.

Data type prefix	Type	Use (memory location)	Example
x	BOOL	boolean (8 bit)	xName
by	BYTE	bit sequence (8 bit)	byName
w	WORD	bit sequence (16 bit)	wName
dw	DWORD	bit sequence (32 bit)	dwName
lw	LWORD	bit sequence (64 bit)	lwName
si	SINT	short integer (8 bit)	siName
i	INT	integer (16 bit)	iName
di	DINT	doubled integer (32 bit)	diName
li	LINT	long integer (64 bit)	liName
uli	ULINT	long integer (64 bit)	uliName
usi	USINT	short integer (8 bit)	usiName
ui	UINT	integer (16 bit)	uiName
udi	UDINT	doubled integer (32 bit)	udiName
xw	__XWORD	bit sequence with 32 or 64 bits, depending on the target system	xwName
uxi	__UXINT	double or long unsigned integer, depending on the target system	uxiName
xi	__XINT	double or long signed integer, depending on the target system	xiName
r	REAL	floating-point number (32 bit)	rName
lr	LREAL	doubled floating-point number (64 bit)	lrName
dat	DATE	date (32 bit)	datName
t	TOD	time (32 bit)	tName
dt	DT	date and time (32 bit)	dtName
tim	TIME	duration (32 bit)	timName
ltim	LTIME	duration (64 bit)	ltimName
s	STRING	character string ASCII	sName
ws	WSTRING	character string unicode	wsName
p	pointers	pointer	pxName
r	reference	reference	rxName
a	array	field	axName
e	enumeration	list type	eName
st	struct	structure	stName
if	interface	interface	ifMotion

Data type prefix	Type	Use (memory location)	Example
ut	union	union	utName
fb	function block	function block	fbName

Examples

```
piCounter: POINTER TO INT;
aiCounters: ARRAY [1..22] OF INT;
paiRefCounter: POINTER TO ARRAY [1..22] OF INT;
apstTest   : ARRAY[1..2] OF POINTER TO ST_MotionStructure;
rdiCounter : REFERENCE TO DINT;
ifMotion   : IF_Motion;
```

Property Prefix

Overview

The property prefix identifies the properties of variables and constants.

Prefix Type	Use	Syntax	Example
c_	VAR CONSTANT	local constant	c_xName
r_	VAR RETAIN	remanent variable type retain	r_xName
p_	VAR PERSISTENT	remanent variable type persistent	p_xName
rp_	VAR PERSISTENT	remanent variable of type retain persistent	rp_xName
i_	VAR_INPUT	input parameter of a POU	i_xName
q_	VAR_OUTPUT	output parameter of a POU	q_xName
iq_	VAR_IN_OUT	in-/output parameter of a POU	iq_xName
ati_	AT %IX x.y AT %IB z AT %IW k	input variable that should write on the IEC input area	ati_x0_0MasterEncoderInitOK
atq_	AT %QX x.y AT %QB z AT %QW k	output variable that should write on the IEC input area	atq_w18AxisNotDone
atm_	AT %MX x.y AT %MB z AT %MW k	marker variable that should write on the IEC marker area	atm_w19ModuleNotReady

NOTE:

- Do not declare constants as RETAIN or PERSISTENT.
- Do not declare any RETAIN variables within POUs. This administers the complete POU in the retain memory area.

Example of AT-Declared Variables

The name of the AT-declared variable also contains the type of the target variable. It is used like the type prefix.

```
ati_xEncoderInit AT %IX0.0 : BOOL;
atq_wAxisNotDone AT %QW18 : WORD;
atm_wModuleNotReady AT %MW19 : WORD;
```

NOTE: A variable can also be allocated to an address in the mapping dialog of a device in the controller configuration (device editor). Whether a device offers this dialog is described in its documentation.

POU Prefix

Overview

The following program organization units (POU) are defined in IEC 61131-3:

- function
- function block
- program
- data structure
- list type
- union
- interface

Additional POUs are offered for the creation and management of test cases:

- test case
- test series
- test resources

The designator is composed of a POU prefix and as short a name as possible (for example, `FB_GetResult`). Just like a variable, use uppercase for the first letter of each word in the name. Write the rest in lower case letters. Form a composite POU name from a verb and a noun.

The prefix is written with an underscore before the name and identifies the type of POU based on the table:

POU prefix	Type	Use	Example
SR_	PROGRAM	program	SR_FlowPackerMachine
FB_	FUNCTION_BLOCK	function blocks	FB_VisuController
FC_	FUNCTION	functions	FC_SetUserFault
ST_	STRUCT	data structure	ST_StandardModuleInterface
ET_	Enumeration	list type	ST_StandardModuleInterface
UT_	UNION	union	UT_Values
IF_	INTERFACE	interface	IF_CamProfile
TC_	Testcase*	test case	TC_MultiCam
TS_	TestSeries*	test series	TS_Robotic
TR_	Test resources*	test resources	TR_Axes
*IEC identifier enhancements introduced by Schneider Electric ETest plugin			

Namespace Prefix

Overview

You can view the namespace of a library in the **Library Manager**. Use a short acronym (PacDriveLib -> `PDL`) as namespace. Do not change the default namespace of a library.

To reserve an unambiguous namespace for your own, self-developed libraries, contact your Schneider Electric responsible.

Example

A function `FC_DoSomething()` is located within the library TestlibraryA (namespace `TLA`) as well as in TestlibraryB (namespace `TLB`). The respective function is accessed by prefixing the namespace.

If both libraries are located within a project, the following call-up results in an error detected during compilation:

```
FC_DoSomething();
```

In this case, it is necessary to define clearly which POU is to be called up.

```
TLA.FC_DoSomething();
TLB.FC_DoSomething();
```

SmartTag Functions

SmartTag Function

Overview

The SmartTag function provides support in writing program code by suggesting commands suitable at the programming element. If the cursor is placed on a programming element with SmartTag function, the light bulb icon  is displayed. Click the  icon to display a list of commands available for this programming element.

Available smart tags:

- The declare variable command (see Menu Commands, Online Help) is available for undeclared variables in the implementation part of the ST editor.

Operators

What's in This Chapter

Arithmetic Operators 678
Bitstring Operators 688
Bit-Shift Operators 691
Selection Operators 695
Comparison Operators 699
Address Operators 703
Calling Operator 705
Type Conversion Operators 706
Numeric Functions 719
IEC Extending Operators 725
Initialization Operator 741
Multicore Operators 742

Overview

All IEC operators are supported. In contrast to the standard functions, these operators are recognized implicitly throughout the project.

Besides the IEC operators, the following operators are supported which are not specified by the standard:

- `ANDN`
- `ORN`
- `XORN`
- `SIZEOF` (refer to arithmetic operators, page 678)
- `ADR`
- `BITADR`
- content operator (refer to address operators, page 703
- some scope operators, page 739
- `__CURRENTTASK`
- `__COMPARE_AND_SWAP`
- `__XADD`
- `__TEST_AND_SET`

Arithmetic Operators

Overview

The following operators, specified by the IEC1131-3 standard, are available:

- `ADD`
- `MUL`
- `SUB`
- `DIV`
- `MOD`
- `MOVE`

Additionally, there is the following standard-extending operator:

- `SIZEOF`

Consider possible overflows of arithmetic operations in case the resulting value exceeds the range of the data type used for the result variable. This may result in low values being written to the machine instead of high values or vice versa.

⚠WARNING

UNINTENDED EQUIPMENT OPERATION

Always verify the operands and results used in mathematical operations to avoid arithmetic overflow.

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

ADD

Overview

IEC operator for the addition of variables

Allowed types

- __UXINT
- __XINT
- __XWORD
- BYTE
- DATE
- DATE_AND_TIME
- DINT
- DT
- DWORD
- INT
- LDATE
- LDATE_AND_TIME
- LDT
- LINT
- LREAL
- LTIME
- LTOD
- LWORD
- REAL
- SINT
- TIME
- TIME_OF_DAY
- TOD
- UDINT
- UINT
- ULINT
- USINT
- WORD

For time data types, the following combinations are possible:

- TIME+TIME=TIME

- TIME+LTIME=LTIME
- LTIME+LTIME=LTIME

For date and time data types, the following combinations are possible:

- TOD+TIME=TOD
- DT+TIME=DT
- TOD+LTIME=LTOD
- DT+LTIME=LDT
- LTOD+TIME=LTOD
- LDT+LTIME=LDT
- LTOD+LTIME=LTOD
- LDT+LTIME=LDT

In the FBD/LD editor, the `ADD` operator is an extensible box. This means, instead of a series of concatenated `ADD` boxes, you can use 1 box with multiple inputs. Use the command **Insert Input** for adding further inputs. The number is unlimited.

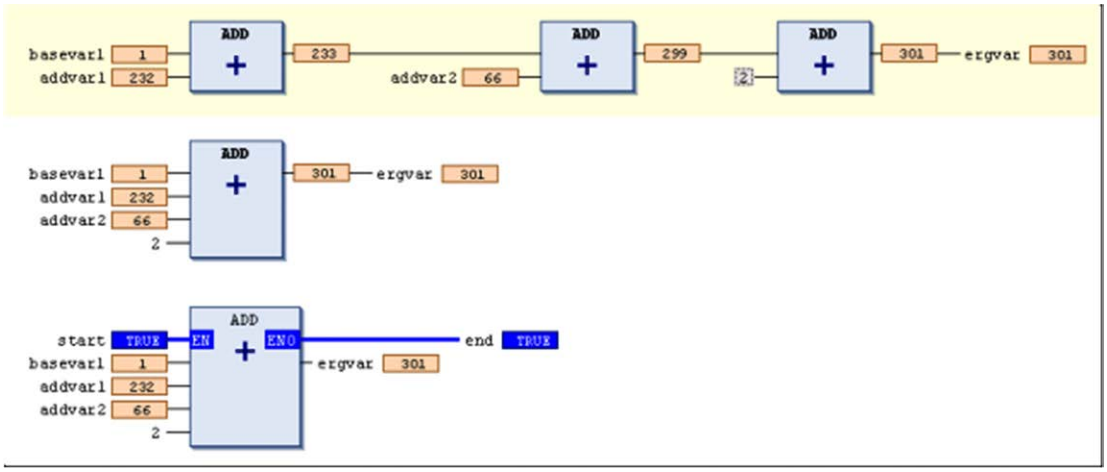
Example in IL

```
LD      7
ADD     2
ADD     4
ADD     7
ST      iVar
```

Example in ST

```
var1 := 7+2+4+7;
```

Examples in FBD



1. series of `ADD` boxes

2. extended `ADD` box

3. `ADD` box with `EN/ENO` parameters

MUL

Overview

IEC operator for the multiplication of variables

Allowed types

- BYTE
- WORD
- DWORD
- LWORD
- SINT
- USINT
- INT
- UINT
- DINT
- UDINT
- LINT
- ULINT
- REAL
- LREAL
- TIME

TIME variables can be multiplied with integer variables.

In the FBD/LD editor, the `MUL` operator is an extensible box. This means, instead of a series of concatenated `MUL` boxes, you can use 1 box with multiple inputs. Use the command **Insert Input** for adding further inputs. The number is unlimited.

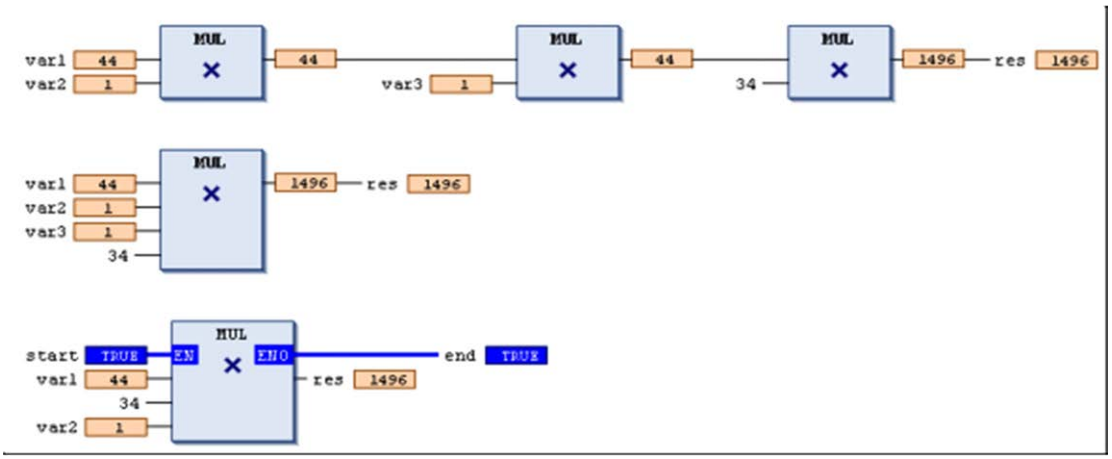
Example in IL

```
LD      7
MUL     2       ,
        4       ,
        7
ST      Var1
```

Example in ST

```
var1 := 7*2*4*7;
```

Examples in FBD



1. series of `MUL` boxes

2. extended `MUL` box

3. `MUL` box with `EN/ENO` parameters

SUB

Overview

IEC operator for the subtraction of one variable from another one.

Allowed types:

- BYTE
- WORD
- DWORD
- LWORD
- SINT
- USINT
- INT
- UINT
- DINT
- UDINT
- LINT
- ULINT
- REAL
- LREAL
- TIME
- LTIME
- TIME_OF_DAY(TOD)
- LTIME_OF_DAY(LTOD)
- DATE
- LDATE
- DATE_AND_TIME(DT)
- LDATE_AND_TIME(LDT)

For time data types, the following combinations are possible:

- TIME–TIME=TIME
- LTIME–LTIME=LTIME

For date and time data types, the following combinations are possible:

- DATE–DATE=TIME
- LDATE–LDATE=LTIME
- TOD–TIME=TOD
- LTOD–TIME=LTOD
- TOD–TOD=TIME
- LTOD–LTOD=LTIME
- DT-TIME=DT
- LDT-LTIME=LDT
- DT-DT=TIME
- LDT–LDT=LTIME

Consider that negative TIME / LTIME values are undefined.

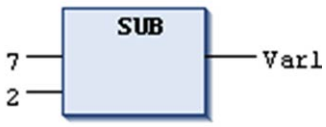
Example in IL

```
LD      7
SUB     2
ST      Var1
```

Example in ST

```
var1 := 7-2;
```

Example in FBD



DIV

Overview

IEC operator for the division of one variable by another one:

Allowed types:

- BYTE
- WORD
- DWORD
- LWORD
- SINT
- USINT
- INT
- UINT
- DINT
- UDINT

- LINT
- ULINT
- REAL
- LREAL
- TIME

TIME variables can be divided by integer variables.

Example in IL

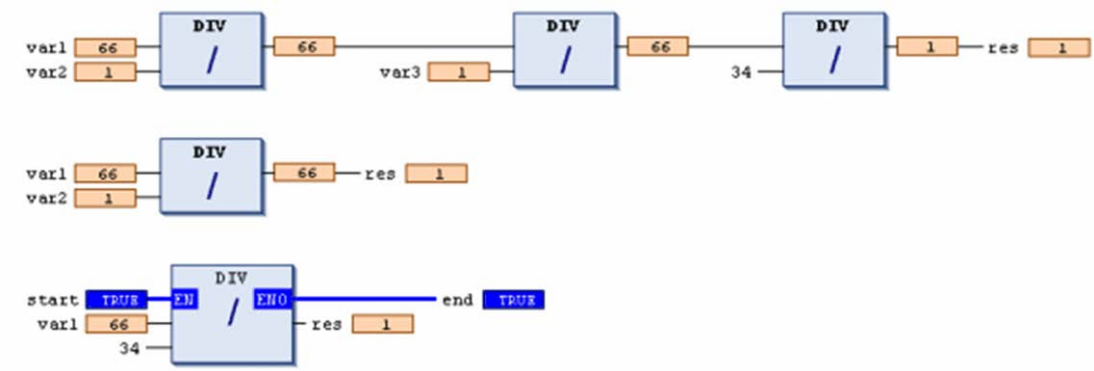
(Result in `Var1` is 4.)

```
LD      8
DIV     2
ST      Var1
```

Example in ST

```
var1 := 8/2;
```

Examples in FBD



1. series of `DIV` boxes

2. single `DIV` box

3. `DIV` box with `EN/ENO` parameters

Different target systems may behave differently concerning a division by zero error. It can lead to a controller HALT, or may go undetected.

⚠WARNING

UNINTENDED EQUIPMENT OPERATION

Use the check functions described in this document, or write your own checks to avoid division by zero in the programming code.

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

NOTE: For more information about the implicit check functions, refer to the chapter *POUs for Implicit Checks*, page 191.

Check Functions

You can use the following check functions to verify the value of the divisor in order to avoid a division by 0 and adapt them, if necessary:

- `CheckDivDInt`
- `CheckDivLint`
- `CheckDivReal`
- `CheckDivLReal`

For information on inserting the function, refer to the description of the **POUs for implicit checks** function, page 191.

The check functions are called automatically before each division found in the application code.

See the following example for an implementation of the function `CheckDivReal`.

Default Implementation of the Function `CheckDivReal`

Declaration part

```
// Implicitly generated code : DO NOT EDIT
FUNCTION CheckDivReal : REAL
VAR_INPUT
 divisor:REAL;
END_VAR
```

Implementation part:

```
// Implicitly generated code : only an suggestion for
implementation
IF divisor = 0 THEN
 CheckDivReal:=1;
ELSE
 CheckDivReal:=divisor;
END_IF;
```

The operator `DIV` uses the output of function `CheckDivReal` as a divisor. In the following example, a division by 0 is prohibited as with the 0 initialized value of the divisor `d` is changed to 1 by `CheckDivReal` before the division is executed. Therefore, the result of the division is 799.

```
PROGRAM PLC_PRG
VAR
 erg:REAL;
 v1:REAL:=799;
 d:REAL;
END_VAR
erg:= v1 / d;
```

MOD

Overview

IEC operator for the modulo division of one variable by another one.

Allowed types:

- BYTE
- WORD
- DWORD
- LWORD
- SINT
- USINT
- INT
- UINT
- DINT
- UDINT

- LINT
- ULINT

The result of this function is the integer remainder of the division.

Different target systems may behave differently concerning a division by zero error. It can lead to a controller HALT, or may go undetected.

⚠WARNING

UNINTENDED EQUIPMENT OPERATION

Use the check functions described in this document, or write your own checks to avoid division by zero in the programming code.

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

NOTE: For more information about the implicit check functions, refer to the chapter *POUs for Implicit Checks*, page 191.

Example in IL

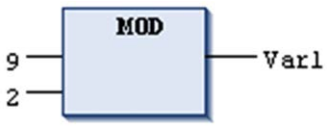
Result in `Var1` is 1.

```
LD      9
MOD     2
ST      Var1
```

Example in ST

```
var1 := 9 MOD 2;
```

Examples in FBD



MOVE

Overview

IEC operator for the assignment of a variable to another variable of an appropriate data type.

The `MOVE` operator is possible for all data types.

As `MOVE` is available as a box in the graphic editors FBD, LD, CFC, there the (unlocking) `EN/ENO` functionality can also be applied on a variable assignment.

Example in CFC in Conjunction with the `EN/ENO` Function

Only if `en_i` is TRUE, `var1` will be assigned to `var2`.



Example in IL

Result: `var2` gets value of `var1`

```
LD      var1
MOVE
ST      var2
```

You get the same result with

```
LD      var1
ST      var2
```

Example in ST

```
ivar2 := MOVE(ivar1);
```

You get the same result with

```
ivar2 := ivar1;
```

SIZEOF

Overview

This arithmetic operator is not specified by the standard IEC 61131-3.

You can use it to determine the number of bytes required by the given variable `x`.

The `SIZEOF` operator returns an unsigned value. The type of the return value will be adapted to the found size of variable `x`.

Return Value of SIZEOF(x)	Data Type of the Constant Implicitly Used for the Found Size
0 ≤ size of x < 256	USINT
256 ≤ size of x < 65,536	UINT
65,536 ≤ size of x < 4,294,967,296	UDINT
4,294,967,296 ≤ size of x	ULINT

NOTE: With EcoStruxure Machine Expert V2.2 and later versions, use the `XSIZEOF` operator, page 688 instead of this `SIZEOF` operator.

Example in ST

```
var1 := SIZEOF(arr1);  (* d.h.: var1:=USINT#10; *)
```

Example in IL

Result is 10

```
arr1:ARRAY[0..4] OF INT;
Var1:INT;
LD      arr1
SIZEOF
ST      Var1
```

XSIZEOF

Overview

This arithmetic operator is not specified by the standard IEC 61131-3.

You can use it to determine the number of bytes required by the passed variable or data type.

The `XSIZEOF` operator returns an unsigned value.

The data type of the return value `<return value>` is:

- ULINT on 64-bit platforms
- UDINT on platforms other than 64-bit

To generate code that runs on all platforms, the return value can be declared with the __UXINT data type.

Syntax

```
<return value> := XSIZEOF( <variable> );
```

Example in ST

```
PROGRAM PLC_PRG
VAR
    uxiReturnValue : __UXINT;  (* data type for 64-bit
platforms: ULINT *)
    aData_1 : ARRAY[0..4] OF INT;
END_VAR
uxiReturnValue := XSIZEOF(aData_1);
```

Result:
`uxiReturnValue = 10`

> **NOTE:** In the case of assignment to a variable of type __UXINT, use the `XSIZEOF` operator instead of the `SIZEOF` operator because with `XSIZEOF`, the data type of the return value depends on the platform.

Bitstring Operators

Overview

The following bitstring operators are available, matching the IEC1131-3 standard:

- AND, page 689
- OR, page 689
- XOR, page 690
- NOT, page 691

The following operators are not specified by the standard and are not available:

- `ANDN`
- `ORN`
- `XORN`

Bitstring operators compare the corresponding bits of 2 or several operands.

AND

Overview

IEC bitstring operator for bitwise AND of bit operands.

If the input bits each are 1, then the resulting bit will be 1, otherwise 0.

Allowed types:

- BOOL
- BYTE
- WORD
- DWORD
- LWORD

Example in IL

Result in `Var1` is 2#1000_0010.

```
Var1:BYTE;
LD      2#1001_0011
AND     2#1000_1010
ST      var1
```

Example in ST

```
var1 := 2#1001_0011 AND 2#1000_1010
```

Example in FBD



OR

Overview

IEC bitstring operator for bitwise OR of bit operands.

If at least 1 of the input bits is 1, the resulting bit will be 1, otherwise 0.

Allowed types:

- BOOL
- BYTE
- WORD
- DWORD
- LWORD

Example in IL

Result in `var1` is 2#1001_1011.

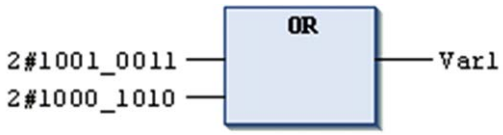
```
var1:BYTE;
LD      2#1001_0011
```

```
OR      2#1000_1010
ST      Var1
```

Example in ST

```
Var1 := 2#1001_0011 OR 2#1000_1010
```

Example in FBD



XOR

Overview

IEC bitstring operator for bitwise XOR of bit operands.

If only 1 of the input bits is 1, then the resulting bit will be 1; if both or none are 1, the resulting bit will be 0.

Allowed types:

- BOOL
- BYTE
- WORD
- DWORD
- LWORD

NOTE: `XOR` allows adding additional inputs. If more than 2 inputs are available, then an XOR operation is performed on the first 2 inputs. The result, in turn, will be XOR combined with input 3, and so on. This has the effect that an odd number of inputs will lead to a resulting bit = 1.

Example in IL

Result is 2#0001_1001.

```
Var1:BYTE;
LD      2#1001_0011
XOR     2#1000_1010
ST      var1
```

Example in ST

```
Var1 := 2#1001_0011 XOR 2#1000_1010
```

Example in FBD



NOT

Overview

IEC bitstring operator for bitwise NOT operation of a bit operand.

The resulting bit will be 1 if the corresponding input bit is 0 and vice versa.

Allowed types

- BOOL
- BYTE
- WORD
- DWORD
- LWORD

Example in IL

Result in `Var1` is 2#0110_1100.

```
Var1:BYTE;
LD      2#1001_0011
NOT
ST      var1
```

Example in ST

```
Var1 := NOT 2#1001_0011
```

Example in FBD



Bit-Shift Operators

SHL

Overview

IEC operator for bitwise left-shift of an operand.

```
erg:= SHL (in, n)
```

`in`: operand to be shifted to the left

`n`: number of bits, by which `in` gets shifted to the left

NOTE: If `n` exceeds the data type width, it depends on the target system how BYTE, WORD, DWORD and LWORD operands will be filled. Some cause filling with zeros (`0`), others with `n MOD <register width>`.

NOTE: The amount of bits which is considered for the arithmetic operation depends on the data type of the input variable. If the input variable is a constant, the smallest possible data type is considered. The data type of the output variable has no effect at all on the arithmetic operation.

Examples

See in the following example in hexadecimal notation the different results for `erg_byte` and `erg_word`. The result depends on the data type of the input variable (BYTE or WORD), although the values of the input variables `in_byte` and `in_word` are the same.

Example in ST

```
PROGRAM shl_st
VAR
 in_byte : BYTE:=16#45; (* 2#01000101 )
 in_word : WORD:=16#0045; (* 2#0000000001000101 )
 erg_byte : BYTE;
 erg_word : WORD;
 n: BYTE :=2;
END_VAR
erg_byte:=SHL(in_byte,n); (* Result is 16#14, 2#00010100 *)
erg_word:=SHL(in_word,n); (* Result is 16#0114,
2#0000000100010100 *)
```

Example in FBD



Example in IL

```
LD      in_byte
SHL     2
ST      erg_byte
```

SHR

Overview

IEC operator for bitwise right-shift of an operand.
`erg:= SHR (in, n)`

`in`: operand to be shifted to the right

`n`: number of bits, by which `in` gets shifted to the right

NOTE: If `n` exceeds the data type width, it depends on the target system how BYTE, WORD, DWORD and LWORD operands will be filled. Some cause filling with zeros (`0`), others with `n MOD <register width>`.

Examples

The following example in hexadecimal notation shows the results of the arithmetic operation depending on the type of the input variable (BYTE or WORD).

Example in ST

```
PROGRAM shr_st
```

```
VAR
 in_byte : BYTE:=16#45; (* 2#01000101 )
 in_word : WORD:=16#0045; (* 2#0000000001000101 )
 erg_byte : BYTE;
 erg_word : WORD;
 n: BYTE :=2;
END_VAR
erg_byte:=SHR(in_byte,n); (* Result is 16#11, 2#00010001 *)
erg_word:=SHR(in_word,n); (* Result is 16#0011,
2#0000000000010001 *)
```

Example in FBD



Example in IL

```
LD      in_byte
SHR     2
ST      erg_byte
```

ROL

Overview

IEC operator for bitwise rotation of an operand to the left.
`erg:= ROL (in, n)`

Allowed data types

- BYTE
- WORD
- DWORD
- LWORD

`in` will be shifted 1 bit position to the left `n` times while the bit that is furthest to the left will be reinserted from the right

> **NOTE:** The amount of bits which is considered for the arithmetic operation depends on the data type of the input variable. If the input variable is a constant, the smallest possible data type is considered. The data type of the output variable has no effect at all on the arithmetic operation.

Examples

See in the following example in hexadecimal notation the different results for `erg_byte` and `erg_word`. The result depends on the data type of the input variable (BYTE or WORD), although the values of the input variables `in_byte` and `in_word` are the same.

Example in ST

```
PROGRAM rol_st
VAR
in_byte : BYTE:=16#45;
in_word : WORD:=16#45;
```

```
erg_byte : BYTE;
erg_word : WORD;
n: BYTE :=2;
END_VAR
erg_byte:=ROL(in_byte,n); (* Result is 16#15 *)
erg_word:=ROL(in_word,n); (* Result is 16#0114 *)
```

Example in FBD



Example in IL

```
LD      in_byte
ROL     n
ST      erg_byte
```

ROR

Overview

IEC operator for bitwise rotation of an operand to the right.
`erg:= ROR (in, n)`

Allowed data types

- BYTE
- WORD
- DWORD
- LWORD

`in` will be shifted 1 bit position to the right `n` times while the bit that is furthest to the left will be reinserted from the left.

NOTE: The amount of bits which are taken into account for the arithmetic operation depends on the data type of the input variable. If the input variable is a constant, the smallest possible data type is taken into account. The data type of the output variable has no effect at all on the arithmetic operation.

Examples

See in the following example in hexadecimal notation the different results for `erg_byte` and `erg_word`. The result depends on the data type of the input variable (BYTE or WORD), although the values of the input variables `in_byte` and `in_word` are the same.

Example in ST

```
PROGRAM ror_st
VAR
in_byte : BYTE:=16#45;
in_word : WORD:=16#45;
erg_byte : BYTE;
erg_word : WORD;
n: BYTE :=2;
END_VAR
```

```
erg_byte:=ROR(in_byte,n); (* Result is 16#51 *)
erg_word:=ROR(in_word,n); (* Result is 16#4011 *)
```

Example in FBD



Example in IL

```
LD      in_byte
ROR     n
ST      erg_byte
```

Selection Operators

Overview

Selection operations can also be performed with variables.

For purposes of clarity the examples provided in this document are limited to the following which use constants as operators:

- SEL, page 695
- MAX, page 697
- MIN, page 697
- LIMIT, page 698
- MUX, page 698

SEL

Overview

IEC selection operator for binary selection.

G determines whether *IN0* or *IN1* is assigned to *OUT*.

`OUT := SEL(G, IN0, IN1)` means:

`OUT := IN0;` if `G`=FALSE

`OUT := IN1;` if `G`=TRUE

Allowed data types:

`IN0`, `IN1`, and `OUT` can be any identical data type. Make sure that variables of the identical data type are used at these positions, especially when using user-defined data types. The compiler verifies the identity of the types and returns compiler errors. Assigning function block instances to interface variables is not supported.

`G`: BOOL

Example in IL

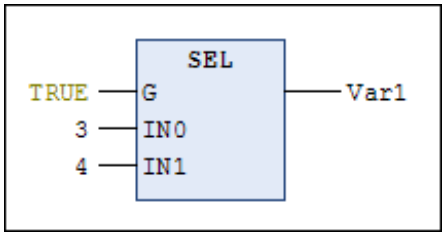
```
LD   TRUE
```

```
SEL 3,4    (* IN0 = 3, IN1 =4 *)
ST  Var1   (* result is 4 *)
LD  FALSE
SEL 3,4
ST  Var1   (* result is 3 *)
```

Example in ST

```
Var1:=SEL(TRUE,3,4); (* result is 4 *)
```

Example in FBD



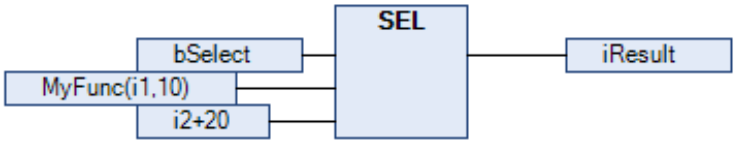
Note

NOTE: When `G` is TRUE, a textual expression that precedes `IN0` is not computed.

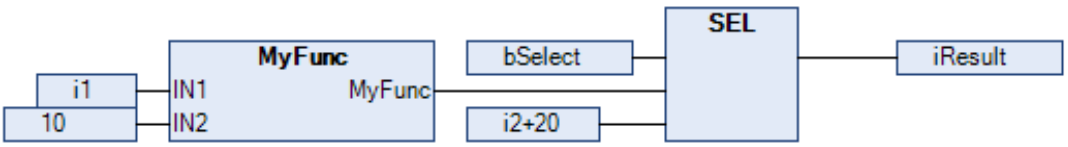
When `G` is FALSE, a textual expression that precedes `IN1` is not computed.

In graphical programming languages, where a Box, Jump, Return, Line Branch, or Edge Detection is connected to `IN0` or `IN1`, these statements will always be computed, independently of the input `G`. This may have an impact on the amount of executed code and thus on the performance. Verify your code carefully to see whether the expressions connected to the inputs `IN0` and `IN1` should always be executed or only depending on the input `G`.

Example 1: `MyFunc` is directly connected to the input `IN1`. `MyFunc` is only executed if `bSelect` at the input `G` is FALSE:



Example 2: `MyFunc` is connected via a link to the input `IN1`. `MyFunc` is always executed (regardless whether `bSelect` at the input `G` is FALSE or TRUE):



MAX

Overview

IEC selection operator performing a maximum function.

The MAX operator returns the greatest value of the inputs.

```
OUT := MAX(IN0, IN1, IN2,...)
```

IN0, IN1, IN2,... and OUT can be any type of variable.

Example in IL

Result is 90

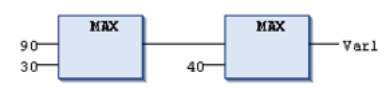
```
LD      90
MAX     30
MAX     40
MAX     77
ST      Var1
```

Example in ST

Result is 90

```
Var1 := MAX(30,90,40);
Var1 := MAX(40,MAX(90,30));
```

Example in FBD



MIN

Overview

IEC selection operator performing a minimum function.

The MIN operator returns the smallest value of the inputs.

```
OUT := MIN(IN0, IN1, IN2,...)
```

IN0, IN1, IN2,... and OUT can be any type of variable.

Example in IL

Result is 30

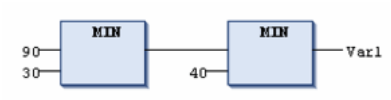
```
LD      90
MIN     30
MIN     40
MIN     77
ST      Var1
```

Example in ST

```
Var1 := MIN(90,30,40);
```

```
Var1 := MIN(MIN(90,30),40);
```

Example in FBD



LIMIT

Overview

IEC selection operator performing a limiting function.
`OUT := LIMIT(Min, IN, Max)` means:
`OUT := MIN (MAX (IN, Min), Max)`

`Max` is the upper and `Min` the lower limit for the result. Should the value `IN` exceed the upper limit `Max`, `LIMIT` will return `Max`. Should `IN` fall below `Min`, the result will be `Min`.

`IN` and `OUT` can be any type of variable.

Example in IL

Result is 80

```
LD      90
LIMIT   30      ,
        80
ST      Var1
```

Example in ST

```
Var1:=LIMIT(30,90,80); (* Result is 80 *);
```

MUX

Overview

IEC selection operator for multiplexing operation.

`OUT := MUX(K, IN0,...,INn))` means:
`OUT := INk`

`IN0, ...,INn` and `OUT` can be any identical data type. Make sure that variables of the identical data type are used at these positions, especially when using user-defined data types. The compiler verifies the identity of the types and returns compiler errors. Assigning function block instances to interface variables is not supported.

`K` has to be BYTE, WORD, DWORD, LWORD, SINT, USINT, INT, UINT, DINT, LINT, ULINT or UDINT.

`MUX` selects the K^{th} value from among a group of values.

Example in IL

Result is 30

```
LD      0
MUX     30      ,
        40      ,
        50      ,
        60      ,
        70      ,
        80
ST      Var1
```

Example in ST

```
Var1:=MUX(0,30,40,50,60,70,80); (* Result is 30 *);
```

NOTE: A textual expression occurring ahead of an input other than `INk` will not be processed.

In graphical programming languages, where a Box, Jump, Return, Line Branch, or Edge Detection is connected to `IN0` or `INn`, these statements will always be computed, independently of the input `K`. This may have an impact on the amount of executed code and thus on the performance. Verify your code carefully to see whether the expressions connected to the inputs `IN0`, `INn` should always be executed or only depending on the input `K`.

For an example illustrating this issue, refer to the `SEL` operator, page 696.

Comparison Operators

Overview

The following operators matching the IEC1131-3 standard are available:

- GT, page 699
- LT, page 700
- LE, page 700
- GE, page 701
- EQ, page 702
- NE, page 702

GT

Overview

Comparison operator performing a Greater Than function.

The `GT` operator is a boolean operator which returns the value TRUE when the value of the first operand is greater than that of the second.

Elementary data types, page 635 are permitted as data types for the operands.

Example in IL

Result is FALSE

```
LD      20
GT      30
ST      Var1
```

Example in ST

```
VAR1 := 20 > 30;
```

Example in FBD



LT

Overview

Comparison operator performing a Less Than function.

The `LT` operator is a boolean operator which returns the value TRUE when the value of the first operand is less than that of the second.

Elementary data types, page 635 are permitted as data types for the operands.

Example in IL

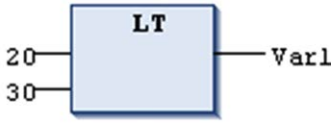
Result is TRUE

```
LD      20
LT      30
ST      Var1
```

Example in ST

```
VAR1 := 20 < 30;
```

Example in FBD



LE

Overview

Comparison operator performing a Less Than Or Equal To function.

The `LE` operator is a boolean operator which returns the value TRUE when the value of the first operand is less than or equal to that of the second.

Elementary data types, page 635 are permitted as data types for the operands.

Example in IL

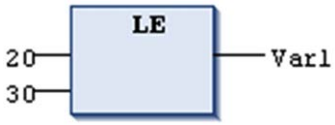
Result is TRUE

```
LD      20
LE      30
ST      Var1
```

Example in ST

```
VAR1 := 20 <= 30;
```

Example in FBD



GE

Overview

Comparison operator performing a Greater Than Or Equal To function.

The `GE` operator is a boolean operator which returns the value TRUE when the value of the first operand is greater than or equal to that of the second.

Elementary data types, page 635 are permitted as data types for the operands.

Example in IL

Result is TRUE

```
LD      60
GE      40
ST      Var1
```

Example in ST

```
VAR1 := 60 >= 40;
```

Example in FBD



EQ

Overview

Comparison operator performing an Equal To function.

The `EQ` operator is a boolean operator which returns the value TRUE when the operands are equal.

Elementary data types, page 635 are permitted as data types for the operands.

Example in IL

Result is TRUE

```
LD      40
EQ      40
ST      Var1
```

Example in ST

```
VAR1 := 40 = 40;
```

Example in FBD



NE

Overview

Comparison operator performing a Not Equal To function.

The `NE` operator is a boolean operator which returns the value TRUE when the operands are not equal.

Elementary data types, page 635 are permitted as data types for the operands.

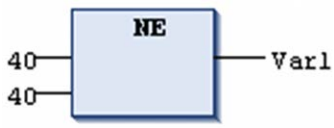
Example in IL

```
LD      40
NE      40
ST      Var1
```

Example in ST

```
VAR1 := 40 <> 40;
```

Example in FBD



Address Operators

ADR

Overview

This address operator is not specified by the standard IEC 61131-3.

ADR returns the address, page 760 of its argument in a DWORD. This address can be assigned to a pointer, page 647 within the project.

NOTE: You can use the ADR operator with function names, program names, function block names, and method names.

Refer to the chapter *Pointers*, page 647 and consider that function pointers can be passed to external libraries. Nevertheless, there is no possibility to call a function pointer within EcoStruxure Machine Expert. In order to enable a system call (runtime system), set the respective object property (in the menu **View > Properties... > Build**) for the function object.

Example in ST

```
VAR
        piAddress1: POINTER TO INT;
        iVar1 : INT := 5;

        lwAddress2 : LWORD;
        iVar2 : INT := 10;

        xwAddress3 : _XWORD;
        iVar3 : INT;
END_VAR
piAddress1 := ADR(iVar1); // piNumber is assigned to address
of iVar1
lwAddress2 := ADR(iVar2); // works for 64 bit run time
system
xwAddress3 := ADR(iVar3); // works for 32 bit and 64 bit
systems
```

Example in IL

```
LD      bVar
ADR
ST      dwVar
```

Considerations for Online Changes

Executing the command **Online Change** can move variables to another place in the memory. There is an indication during online change if copying is necessary.

The shift of variables may have the effect that `POINTER TO` variables point to invalid memory.

⚠WARNING

UNINTENDED EQUIPMENT OPERATION

Assign the value of any `POINTER TO` type variable(s) prior to the first use of it within a POU, and at every subsequent cycle.

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

NOTE: `POINTER TO` variables of functions and methods should not be returned to the caller of this function or passed to global variables.

Content Operator

Overview

This address operator is not specified by the standard IEC 61131-3. You can dereference a pointer by adding the content operator `^` (ASCII caret or circumflex symbol) after the pointer identifier.

Example in ST

```
pt:POINTER TO INT;
var_int1:INT;
var_int2:INT;
pt := ADR(var_int1);
var_int2:=pt^;
```

Considerations for Online Changes

Executing the **Online Change** command can change the contents of addresses.

⚠CAUTION

INVALID POINTER

Verify the validity of the pointers when using pointers on addresses and executing the Online Change command.

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

BITADR

Overview

This address operator is not specified by the standard IEC 61131-3.

`BITADR` returns the bit offset within the segment in a DWORD. The offset value depends on whether the option **Byte addressing** in the target settings is activated or not.

The highest nibble in that DWORD indicates the memory area:

Memory: 16#40000000

Input: 16#80000000

Output: 16#C0000000

Example in ST

```
VAR
        xVar AT %IX2.3:BOOL;
        dwBitoffset: DWORD;
END_VAR
dwBitoffset:=BITADR(xVar); (* Result if byte addressing=
TRUE: 16#80000013, if byte addressing=FALSE: 16#80000023 *)
```

Example in IL

```
LD      xVar
BITADR
ST      dwBitoffset
```

Considerations for Online Changes

Executing the **Online Change** command can change the contents of addresses.

⚠CAUTION

INVALID POINTER

Verify the validity of the pointers when using pointers on addresses and executing the Online Change command.

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

Calling Operator

CAL

Overview

IEC operator for calling a function block or a program.

Use `CAL` in IL to call up a function block instance. Place the variables that will serve as the input variables in parentheses right after the name of the function block instance.

Example

Calling up the instance `Inst` of a function block where input variables `Par1` and `Par2` are 0 and TRUE, respectively.

```
CAL INST(PAR1 := 0, PAR2 := TRUE)
```

Type Conversion Operators

Type Conversion Functions

Overview

It is not allowed to convert implicitly from a larger type to a smaller type (for example, from INT to BYTE or from DINT to WORD). To achieve this, you have to perform special type conversions. You can basically convert from any elementary type to any other elementary type.

Syntax

Typed conversion: `<elem.type1>_TO_<elem.type2>`

Overloaded conversion: `TO_<elem.type2>`

If the operand value for a type conversion operator is outside the value range of the target data type, the result output depends on the controller processor type and is therefore undefined. This is the case, for example, when a negative operand value is converted from LREAL to the target data type UINT.

NOTICE
UNDEFINED OUTPUT Verify that the values of the target data type are within the expected range after type conversion. **Failure to follow these instructions can result in equipment damage.**

When converting the data type to STRING or WSTRING, the typed value is left-aligned as a character string and truncated if it exceeds the length of the target string variable.

NOTICE
STRING MANIPULATION Declare a return variable of sufficient size for the type conversion operators `<>_TO_STRING` and `<>_TO_WSTRING`. **Failure to follow these instructions can result in equipment damage.**

The following type conversions are supported:

- Overloaded conversions, page 707
- BOOL_TO conversions, page 708
- TO_BOOL conversions, page 709
- conversion between integral number types, page 711
- REAL_TO-/ LREAL_TO conversions, page 711
- TIME_TO/TIME_OF_DAY conversions, page 712
- DATE_TO/DT_TO conversions, page 714
- STRING_TO conversions, page 715
- TRUNC, page 716 (conversion to DINT)
- TRUNC_INT, page 717
- ANY_NUM_TO_<numeric data type>

Overloaded Conversions

General Information

For general hints to be considered during type conversion, refer to the chapter *Type Conversion Functions*, page 706.

NOTE: Conversions from larger data types to smaller types may result in loss of information.

Overloaded conversions are not specified by the standard {(notrans) IEC 61131-3}. To comply with the standard, use the operators of the syntax `<type> _TO_ <another type>` as described in this chapter.

The rules for typed conversions also apply to overloaded conversions.

Definition

In overloaded conversions, the operators convert values into other data types, explicitly specifying only a target data type and no source data type (data type of the operands).

Syntax

```
<variable name> := <TO operator> ( <operand> );
<operand> = <variable name> | <literal>
```

Examples in ST

```
VAR
    iNumber_1 : INT;
    rNumber_2 : REAL := 123.456;
    iNumber_2 : INT;
    xIsTrue : BOOL;
    sOutputText : STRING;
    sText : STRING := 'Hello World!';
    wsText: WSTRING;
    dateEvent : DATE := D#2019-9-3;
    uiEvent : UINT;
    uxiData : __UXINT;
END_VAR
```

Examples in ST with conversion results:

Example	Result
`iNumber_1 := TO_INT(4.22);`	4
`iNumber_2 := TO_INT(rNumber_2);`	123
`xIsTrue := TO_BOOL(1);`	TRUE
`sOutputText := TO_STRING(342);`	'342’
`wsText := TO_WSTRING(sText);`	"Hello World!"
`uiEvent := TO_UINT(dateEvent);`	44288
`uxiData := TO___UXINT(iNumber_2);`	123

BOOL_TO Conversions

General Information

For general hints to be considered during type conversion, refer to the chapter *Type Conversion Functions*, page 706.

Definition

IEC operator for conversions from type BOOL to any other type.

Syntax

BOOL_TO_<data type>

Conversion Results

The conversion results for number types and for string types depend on the state of the operand:

Operand State	Result for Number Types	Result for String Types
TRUE	1	TRUE
FALSE	0	FALSE

NOTE: The operators that convert a value into a character string of type STRING or WSTRING require an operand that matches the target data type.

Examples in ST

Examples in ST with conversion results:

Example	Result
`i:=BOOL_TO_INT(TRUE);`	`1`
`str:=BOOL_TO_STRING(TRUE);`	`'TRUE'`
`t:=BOOL_TO_TIME(TRUE);`	`T#1ms`
`tof:=BOOL_TO_TOD(TRUE);`	`TOD#00:00:00.001`
`dat:=BOOL_TO_DATE(FALSE);`	`D#1970`
`dandt:=BOOL_TO_DT(TRUE);`	`DT#1970-01-01-00:00:01`

Examples in IL

Examples in IL with conversion results:

Example	Result

`LD TRUE`
`BOOL_TO_INT`
`ST i`
	`1`

`LD TRUE`
`BOOL_TO_STRI...`
`ST str`
	`'TRUE'`

`LD TRUE`
`BOOL_TO_TIME`
`ST t`
	`T#1ms`

Example		Result
LD BOOL_TO_TOD ST	TRUE tof	TOD#00:00:00.001
LD BOOL_TO_DATE ST	FALSE dandt	D#1970-01-01
LD BOOL_TO_DT ST	TRUE dandt	DT#1970-01-01-00:00:01

Examples in FBD

Examples in FBD with conversion results:

Example	Result
	1
	'TRUE'
	T#1ms
	TOD#00:00:00.001
	D#1970-01-01
	DT#1970-01-01-00:00:01

TO_BOOL Conversions

General Information

For general hints to be considered during type conversion, refer to the chapter *Type Conversion Functions*, page 706.

Definition

IEC operator for conversions from another variable type to BOOL.

Syntax

<data type>_TO_BOOL

Conversion Results

The result is TRUE when the operand is not equal to 0. The result is FALSE when the operand is equal to 0.

The result is TRUE for STRING type variables when the operand is TRUE. Otherwise the result is FALSE.

Examples in ST

Examples in ST with conversion results:

Example	Result
`b := BYTE_TO_BOOL(2#11010101);`	TRUE
`b := INT_TO_BOOL(0);`	FALSE
`b := TIME_TO_BOOL(T#5ms);`	TRUE
`b := STRING_TO_BOOL('TRUE');`	TRUE

Examples in IL

Examples in IL with conversion results:

Example	Result

`LD 213`
`BYTE_TO_BOOL`
`ST b`
	TRUE

`LD 0`
`INT_TO_BOOL`
`ST b`
	FALSE

`LD T#5ms`
`TIME_TO_BOOL`
`ST b`
	TRUE

`LD 'TRUE'`
`STRING_TO_BOOL`
`ST b`
	TRUE

Examples in FBD

Examples in FBD with conversion results:

Example	Result
	TRUE
	FALSE
	TRUE
	TRUE

Conversion Between Integral Number Types

General Information

For general hints to be considered during type conversion, refer to the chapter *Type Conversion Functions*, page 706.

Definition

Conversion from an integral number type to another number type.

Syntax

<INT data type>_TO_<INT data type>

For information on the integer data type, refer to the chapter *Standard Data Types*, page 636.

Conversion Results

If the number you are converting exceeds the range limit, the first bytes for the number will be ignored.

Example in ST

```
si := INT_TO_SINT(4223); (* Result is 127 *)
```

If you save the integer 4223 (16#107f represented hexadecimally) as a SINT variable, it will appear as 127 (16#7f represented hexadecimally).

Example in IL

```
LD                  4223
INT_TO_SINT
ST                  si
```

Example in FBD



REAL_TO / LREAL_TO Conversions

General Information

For general hints to be considered during type conversion, refer to the chapter *Type Conversion Functions*, page 706.

Definition

IEC operator for conversions from the variable type REAL or LREAL to a different type.

The value will be rounded up or down to the nearest whole number and converted into the new variable type.

Exceptions to this are the following variable types:

- STRING
- BOOL
- REAL
- LREAL

NOTE: The rounding logic that is applied depends on the target system or the FPU (Floating Point unit) of the target system. A value of `-1.5` can thus be converted differently on different controllers.

Conversion Results

If a REAL or LREAL is converted to SINT, USINT, INT, UINT, DINT, UDINT, LINT or ULINT and the value of the real number is out of the value range of that integer, the result will be undefined, and may lead to a controller exception.

NOTE: Validate any range overflows by your application and verify that the value of the REAL or LREAL is within the bounds of the target integer before performing the conversion.

When converting to type STRING, consider that the total number of digits is limited to six. If the (L)REAL number has more digits, then the sixth will be rounded. If the length of the STRING is defined too short, it will be cut from the right end.

Example in ST

Examples in ST with conversion results:

Example	Result
`i := REAL_TO_INT(1.5);`	`2`
`j := REAL_TO_INT(1.4);`	`1`
`i := REAL_TO_INT(-1.5);`	`-2`
`j := REAL_TO_INT(-1.4);`	`-1`

Example in IL

```
LD                  2.75
REAL_TO_INT
ST                  i
```

Example in FBD



TIME_TO/TIME_OF_DAY Conversions

General Information

For general hints to be considered during type conversion, refer to the chapter *Type Conversion Functions*, page 706.

Definition

IEC operator for conversions from the variable type TIME or TIME_OF_DAY to a different type.

Syntax

TIME_TO_<data type>

TOD_TO_<data type>

Conversion Results

The time will be stored internally in a DWORD in milliseconds (beginning with 12:00 A.M. for the TIME_OF_DAY variable). This value will then be converted.

In case of type STRING the result is a time constant.

NOTE: The operators that convert a value into a character string of type STRING or WSTRING require an operand that matches the target data type.

Examples in ST

Examples in ST with conversion results:

Example	Result
`str := TIME_TO_STRING(T#12ms);`	`'T#12ms'`
`dw := TIME_TO_DWORD(T#5m);`	`300000`
`si := TOD_TO_SINT(TOD#00:00:00.012);`	`12`

Examples in IL

Examples in IL with conversion results:

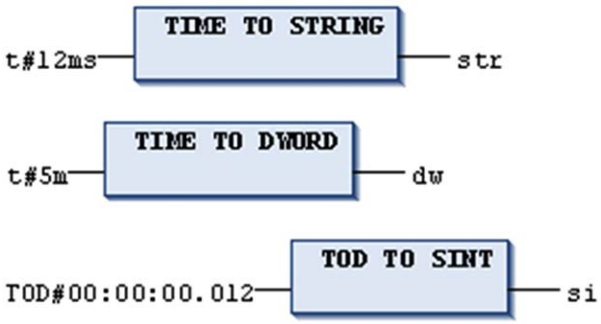
Example	Result

`LD T#12ms`
`TIME_TO_STRI...`
`ST str`
	`'T#12ms'`

`LD T#300000ms`
`TIME_TO_DWORD`
`ST dw`
	`300000`

`LD TOD#00:00:00.012`
`TIME_TO_SINT`
`ST si`
	`12`

Examples in FBD



DATE_TO/DT_TO Conversions

General Information

For general hints to be considered during type conversion, refer to the chapter *Type Conversion Functions*, page 706.

Definition

IEC operator for conversions from the variable type DATE or DATE_AND_TIME to a different type.

Syntax

DATE_TO_<data type>

DT_TO_<data type>

Conversion Results

The date will be stored internally in a DWORD in seconds since Jan. 1, 1970. This value will then be converted.

For STRING type variables, the result is the date constant.

NOTE: The operators that convert a value into a character string of type STRING or WSTRING require an operand that matches the target data type.

Examples in ST

Examples in ST with conversion results:

Example	Result
`b := DATE_TO_BOOL(D#1970-01-01);`	`FALSE`
`i := DATE_TO_INT(D#1970-01-15);`	`29952`
`byt := DT_TO_BYTE(DT#1970-01-15-05:05:05);`	`129`
`str := DT_TO_STRING(DT#1998-02-13-14:20);`	`'DT#1998-02-13-14:20'`

Examples in IL

Examples in IL with conversion results:

Example	Result

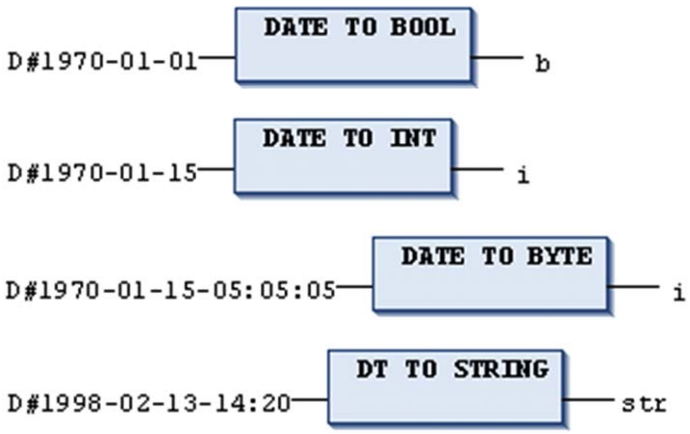
`LD D#1970-01-01`
`DATE_TO_BOOL`
`ST b`
	`FALSE`

`LD D#1970-01-01`
`DATE_TO_INT`
`ST i`
	`29952`

`LD D#1970-01-15-05:05:`
`DATE_TO_BYTE`
`ST byt`
	`129`

`LD D#1998-02-13-14:20`
`DATE_TO_STRI...`
`ST str`
	`'DT#1998-02-13-14:20'`

Examples in FBD



STRING_TO Conversions

General Information

For general hints to be considered during type conversion, refer to the chapter *Type Conversion Functions*, page 706.

Definition

IEC operator for conversions from the variable type STRING to a different type.

Syntax

STRING_TO_<data type>

Specifying Values

Specify the operand of type STRING matching the IEC61131-3 standard. The value must correspond to a valid constant (literal), page 574 of the target type.

Convertible strings can contain:

- Number with type prefix (example: `'16#FFFFFFFF'`)
- Number with grouping characters (example: `'2#1111_1111'`)

  NOTE: Use the underscore, as the international weight and measure grouping character (thin space) is not permitted.
- Floating-point number, also in exponential notation (example: `'9.876'` or `'1.2E-34'`)

  NOTE: Floating-point numbers are not convertible. The comma is treated as a following character and therefore truncated.
- Time, time of day, and date specification with prefix and size (example: `'T#2h'`, `'DT#2019-9-9-12:30:30.9'`)
- Infinite values (example: `'1.7E+400'`)
- Additional character after a number (example: `'2m'` or `'3.14'`) are truncated. Additional characters before a number are not permitted.
- Spaces before (example: `' 3.14'`)

The operand must represent a valid value of the target data type.

NOTE: If the data type of the operand does not match the target type, or if the value exceeds the range of the target data type, then the result depends on the processor type and is therefore undefined.

Conversions from larger types to smaller types may result in loss of information.

⚠CAUTION

LOSS OF DATA

When converting mismatched data types or when the value being converted is larger than the target data type, be sure that the result is validated within your application.

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

Example in IL

Example	Conversion result
LD 'TRUE' STRING_TO_BOOL ST b	TRUE

Examples in ST

Example	Conversion result
b := STRING_TO_BOOL('TRUE');	TRUE
w := STRING_TO_WORD('abc34');	0
w := STRING_TO_WORD('34abc');	34
t := STRING_TO_TIME('T#127ms');	T#127ms
r := STRING_TO_REAL('1.234');	1.234
bv := STRING_TO_BYTE('500');	244

Example in FBD

Example	Conversion result
	TRUE

TRUNC

General Information

For general hints to be considered during type conversion, refer to the chapter *Type Conversion Functions*, page 706.

Definition

IEC operator for conversions from REAL to DINT. The whole number portion of the value will be used.

The result of these functions is not defined if the input value cannot be represented with a DINT or INT. The behavior for such input values is platform-dependent.

Examples in ST

Examples in ST with conversion results:

Example	Result
`diVar:=TRUNC(1.9);`	`1`
`diVar:=TRUNC(-1.4);`	`-1`

Example in IL

```
LD                  1.9
TRUNC
ST                  diVar
```

TRUNC_INT

General Information

For general hints to be considered during type conversion, refer to the chapter *Type Conversion Functions*, page 706.

Definition

IEC operator for conversions from REAL to INT. The whole number portion of the value will be used.

The result of these functions is not defined if the input value cannot be represented with an INT. The behavior for such input values is platform-dependent.

Examples in ST

Examples in ST with conversion results:

Example	Result
`iVar:=TRUNC_INT(1.9);`	`1`
`iVar:=TRUNC_INT(-1.4);`	`-1`

Example in IL

```
LD                  1.9
TRUNC_INT
ST                  iVar
```

ANY_...TO Conversions

General Information

For general hints to be considered during type conversion, refer to the chapter *Type Conversion Functions*, page 706.

Definition

Conversion from any data type, or more specifically from any numeric data type to another data type. As with any type of conversion, the size of the operands must be taken into account in order to have a successful conversion.

Syntax

ANY_NUM_TO_<numeric data type>

ANY_TO_<any data type>

Example

Conversion from a variable of data type REAL to INT:

```
re : REAL := 1.234;
i : INT := ANY_TO_INT(re)
```

TO_<xxx> Conversions

General Information

For general hints to be considered during type conversion, refer to the chapter *Type Conversion Functions*, page 706.

This operator is not specified by the standard IEC 61131-3.

Definition

Conversion of variables from one type to another type. The input type must not be specified explicitly (overloaded conversion).

Also refer to the description of type conversion functions, page 706.

Syntax

```
TO_<data type>
```

Example in ST

Conversion from a variable of data type REAL to INT:

```
VAR
   iVar: INT;
   bVar: BOOL;
   strVar: STRING;
   rVar: REAL;
END_VAR
iVar := TO_INT(4.22);         (* Result:  4 *)
```

```
bVar := TO_BOOL(1);          (* Result: TRUE *)
sVar := TO_STRING(342);      (* Result: '342' *)
rVar := TO_WORD('123');      (* Result: 123 *)
```

Numeric Functions

Overview

This chapter describes the available numeric IEC operators specified by the standard IEC 61131-3.

ABS

Definition

Numeric IEC operator for returning the absolute value of a number.

In- and output can be of any numeric data type.

Example in IL

The result in `i` is 2.

```
LD                  -2
ABS
ST                  i
```

Example in ST

```
i:=ABS(-2);
```

Example in FBD



SQRT

Definition

Numeric IEC operator for returning the square root of a number.

The input variable can be of any numeric data type, the output variable has to be type REAL or LREAL.

Example in IL

The result in `q` is 4.

```
LD                  16
SQRT
ST                  q
```

Example in ST

```
q:=SQRT(16);
```

Example in FBD



LN

Definition

Numeric IEC operator for returning the natural logarithm of a number.

The input variable can be of any numeric data type, the output variable has to be type REAL or LREAL.

Example in IL

The result in `q` is 3.80666.

```
LD                  45
LN
ST                  q
```

Example in ST

```
q:=LN(45);
```

Example in FBD



LOG

Definition

Numeric IEC operator for returning the logarithm of a number in base 10.

The input variable can be of any numeric data type, the output variable has to be type REAL or LREAL.

Example in IL

The result in `q` is 2.49762.

```
LD                  314.5
LOG
ST                  q
```

Example in ST

```
q:=LOG(314.5);
```

Example in FBD



EXP

Definition

Numeric IEC operator for returning the exponential function.

The input variable can be of any numeric data type, the output variable has to be type REAL or LREAL.

Example in IL

The result in q is 7.389056099.

```
LD                  2
EXP
ST                  q
```

Example in ST

```
q:=EXP(2);
```

Example in FBD



SIN

Definition

Numeric IEC operator for returning the sine of an angle.

The input defining the angle in radians and the output variable must be of type REAL or LREAL. The output type must correspond to the input type. Otherwise an error message will be displayed.

> **NOTE:** The value range for the input value is $-2^{63}...+2^{63}$ which does not exactly correspond to the LREAL space of 64 bit (also refer to REAL / LREAL, page 637). If the input value is beyond this range, the function returns the input value.

Example in IL

The result in q is 0.479426.

```
LD                  0.5
SIN
ST                  q
```

Example in ST

```
q:=SIN(0.5);
```

Example in FBD



COS

Definition

Numeric IEC operator for returning the cosine of an angle.

The input defining the angle in arch minutes and the output variable must be of type REAL or LREAL. The output type must correspond to the input type. Otherwise an error message will be displayed.

> **NOTE:** The value range for the input value is $-2^{63}...+2^{63}$ which does not exactly correspond to the LREAL space of 64 bit (also refer to REAL / LREAL, page 637). If the input value is beyond this range, the function returns the input value.

Example in IL

The result in q is 0.877583.

```
LD                  0.5
COS
ST                  q
```

Example in ST

```
q:=COS(0.5);
```

Example in FBD



TAN

Definition

Numeric IEC operator for returning the tangent of a number. The value is calculated in arch minutes.

The input variable can be of any numeric data type where the output variable has to be type REAL or LREAL.

Example in IL

The result in `q` is 0.546302.

```
LD                  0.5
TAN
ST                  q
```

Example in ST

```
q:=TAN(0.5);
```

Example in FBD



ASIN

Definition

Numeric IEC operator for returning the arc sine (inverse function of sine) of a number. The value is calculated in arch minutes.

The input variable can be of any numeric data type where the output variable has to be type REAL or LREAL.

Example in IL

The result in `q` is 0.523599.

```
LD                  0.5
ASIN
ST                  q
```

Example in ST

```
q:=ASIN(0.5);
```

Example in FBD



ACOS

Definition

Numeric IEC operator for returning the arc cosine (inverse function of cosine) of a number. The value is calculated in arch minutes.

The input variable can be of any numeric data type where the output variable has to be type REAL or LREAL.

Example in IL

The result in `q` is 1.0472.

```
LD                  0.5
ACOS
ST                  q
```

Example in ST

```
q:=ACOS(0.5);
```

Example in FBD



ATAN

Definition

Numeric IEC operator for returning the arc tangent (inverse function of tangent) of a number. The value is calculated in arch minutes.

The input variable can be of any numeric data type where the output variable has to be type REAL or LREAL.

Example in IL

The result in `q` is 0.463648.

```
LD                  0.5
ATAN
ST                  q
```

Example in ST

```
q:=ATAN(0.5);
```

Example in FBD



EXPT

Definition

Numeric IEC operator for exponentiation of a variable with another variable:

OUT = *IN1* to the *IN2*

The input variable can be of numeric data types (SINT, USINT, INT, UINT, DINT, UDINT, LINT, ULINT, REAL, LREAL, BYTE, WORD, DWORD, and LWORD) where the output variable has to be type REAL or LREAL.

The result of this function is not defined if the following applies:

- The base is negative.
- The base is zero and the exponent is ≤ 0.

The behavior for such input values is platform-dependent.

Example in IL

The result is 49.

```
LD                  7
EXPT                2
ST                  Var1
```

Example in ST

```
var1:=EXPT(7,2);
```

Example in FBD



IEC Extending Operators

IEC Extending Operators

Overview

In addition to the IEC operators, the following IEC extending operators are supported:

- ADR, page 703
- BITADR, page 704
- SIZEOF, page 687
- __DELETE, page 726
- __ISVALIDREF, page 727
- __NEW, page 728
- __QUERYINTERFACE, page 730
- __QUERYPOINTER, page 731
- '__POSITION', page 737

- ‘‘__POUNAME’’, page 738
- scope operators, page 739

__DELETE

Definition

This operator is not specified by the IEC 61131-3 standard.

For compatibility information, refer to the EcoStruxure Machine Expert/CODESYS compiler version mapping table in the corresponding Compatibility and Migration User Guide.

The __DELETE operator deallocates the memory for objects allocated before via the __NEW operator, page 728.

__DELETE has no return value and its operand will be set to 0 after the operation.

Activate the option **Use dynamic memory allocation** in the **Application build options** view (see Menu Commands, Online Help) (**View > Properties... > Application build options**). Consult the *Programming Guide* specific to your controller for whether the options are available on your controller.

Syntax

__DELETE (<pointer>)

If pointer is a pointer to a function block, the dedicated method *FB_Exit* will be called before the pointer is set to NULL.

> **NOTE:** Use the exact data type of the derived function block and not that of the base function block. Do not use a variable of type POINTER TO BaseFB. This is necessary because if the base function block implements no FB_Exit function, then at the later usage of __DELETE(pBaseFB), no FB_Exit is called.
>
> **NOTE:** Make sure that __DELETE is not called by two tasks simultaneously. It is a good practice to use __DELETE in only one task. If this is not feasible, use a semaphore (SysSemEnter) or a comparable method, to help avoid concurrent calls of __DELETE.

You can use a semaphore (SysSemEnter) to help prevent two tasks from deallocating memory at the same time. In this case, the extensive use of __DELETE causes higher jitter.

Example

In the following example, the function block `FBDynamic` is allocated dynamically via `_NEW` from the POU `PLC_PRG`. By doing so, `FB_Init` method will be called, in which a type `DUT` is allocated. When `__DELETE` is called on, the function block pointer from `PLC_PRG`, `FB_Exit` will be called, which in turn frees the allocated internal type.

```
FUNCTION_BLOCK FBDynamic
VAR_INPUT
in1, in2 : INT;
END_VAR
VAR_OUTPUT
out : INT;
END_VAR
VAR
test1 : INT := 1234;
_inc : INT := 0;
_dut : POINTER TO DUT;
END_VAR
out := in1 + in2;

METHOD FB_Exit : BOOL
VAR_INPUT
bInCopyCode : BOOL;
END_VAR
__Delete(_dut);

METHOD FB_Init : BOOL
VAR_INPUT
bInitRetains : BOOL;
bInCopyCode : BOOL;
END_VAR
_dut := __NEW(DUT);

METHOD INC : INT
VAR_INPUT
END_VAR
_inc := _inc + 1;
INC := _inc;

PLC_PRG(PRG)
VAR
pFB : POINTER TO FBDynamic;
bInit: BOOL := TRUE;
bDelete: BOOL;
loc : INT;
END_VAR
IF (bInit) THEN
pFB := __NEW(FBDynamic);
bInit := FALSE;
END_IF
IF (pFB <> 0) THEN
pFB^(in1 := 1, in2 := loc, out => loc);
pFB^.INC();
END_IF
IF (bDelete) THEN
__DELETE(pFB);
END_IF
```

__ISVALIDREF

Definition

This operator is not specified by the IEC 61131-3 standard.

It allows you to check whether a reference points to a valid value.

For how to use and an example, see the description of the reference, page 646 data type.

__NEW

Definition

This operator is not specified by the IEC 61131-3 standard.

The operator __NEW allocates dynamic memory for function block instances, user-defined data types or arrays of standard data types. The operator returns a suitably typed pointer to the object. If the operator is not used within an assignment, a message is displayed.

Activate the option **Use dynamic memory allocation** in the **Application build options** view (see Menu Commands, Online Help) (**View > Properties... > Application build options**) to use the __NEW operator. Consult the *Programming Guide* specific to your controller for whether the options are available on your controller.

If no memory could be allocated, by __NEW will return 0.

For deallocating, use __DELETE.

Syntax

<pointer name> := __NEW (<type>[,<size>]);

The operator creates a new object of the specified type <type> and returns a pointer to that <type>. The initialization of the object is called after creation. If 0 is returned, the operation has not been completed successfully.

> **NOTE:** Use the exact data type of the derived function block and not that of the base function block. Do not use a variable of type POINTER TO BaseFB. This is necessary because if the base function block implements no FB_Exit function, then at the later usage of __DELETE(pBaseFB), no FB_Exit is called.

If <type> is scalar, the optional operand <size> has to be set additionally and the operator creates an array of scalar types with the size length.

Example

```
pScalarType := __New(ScalarType, length);
```

NOTE: A function block or a user-defined data type created dynamically with *__NEW* has a fixed memory area. The data layout cannot be modified by an online change. Therefore, new variables cannot be added to or deleted from the type, and types cannot be modified.

Therefore, only function blocks out of libraries (because they cannot change) and function blocks with attribute `enable_dynamic_creation` are allowed for the `__NEW` operator. If a function block changes with this flag so that copy code will be necessary, a message is produced.

NOTE: The code for memory allocation needs to be non-re-entrant.

A semaphore (`SysSemEnter`) is used to avoid 2 tasks try to allocate memory at the same time. Thus, extensive usage of `__New` may produce higher jitter.

Example with a scalar type:

```
{attribute 'enable_dynamic_creation'}
TYPE DUT :
    STRUCT
        a,b,c,d,e,f : INT;
    END_STRUCT
END_TYPE
PROGRAM PLC_PRG
VAR
    pDut : POINTER TO DUT;
    bInit: BOOL := TRUE;
    bDelete: BOOL;
END_VAR
IF (bInit) THEN
    pDut := __NEW(DUT);
    bInit := FALSE;
END_IF
IF (bDelete) THEN
    __DELETE(pDut);
END_IF
```

Example with a function block:

```
{attribute 'enable_dynamic_creation'}
FUNCTION_BLOCK FBDynamic
VAR_INPUT
    in1, in2 : INT;
END_VAR
VAR_OUTPUT
    out : INT;
END_VAR
VAR
END_VAR
out := in1 + in2;
PROGRAM_PLC_PRG
VAR
    pFB : POINTER TO FBDynamic;
    loc : INT;
    bInit: BOOL := TRUE;
    bDelete: BOOL;
END_VAR
IF (bInit) THEN
    pFB := __NEW(FBDynamic);
    bInit := FALSE;
END_IF
IF (pFB <> 0) THEN
    pFB^(in1 := 1, in2 := loc, out => loc);
END_IF
IF (bDelete) THEN
    __DELETE(pFB);
END_IF
```

Example with an array:

```
PLC_PRG(PRG)
VAR
    bInit: BOOL := TRUE;
    bDelete: BOOL;
    pArrayBytes : POINTER TO BYTE;
    test: INT;
    parr : POINTER TO BYTE;
END_VAR
IF (bInit) THEN
    pArrayBytes := __NEW(BYTE, 25);
    bInit := FALSE;
END_IF
IF (pArrayBytes <> 0) THEN
    pArrayBytes[24] := 125;
    test := pArrayBytes[24];
END_IF
IF (bDelete) THEN
    __DELETE(pArrayBytes);
END_IF
```

__QUERYINTERFACE

Definition

This operator is not specified by the IEC 61131-3 standard.

At runtime, __QUERYINTERFACE is enabling a type conversion of an interface reference to another. The operator returns a result with type BOOL. TRUE implies, that the conversion is successfully executed.

Syntax

__QUERYINTERFACE(<ITF_Source>, < ITF_Dest>

The operator needs as the first operand an interface reference or a function block instance of the intended type and as second operand an interface reference. After execution of __QUERYINTERFACE, the *ITF_Dest* holds a reference to the intended interface if the object referenced from *ITF_Source* implements the interface. In this case, the conversion is successful and the result of the operator returns TRUE. In all other cases, the operator returns FALSE.

A precondition for an explicit conversion is that not only the *ITF_Source* but also *ITF_Dest* is an extension of the interface *__System.IQueryInterface*. This interface is provided implicitly and needs no library.

Example

Example in ST:

```
INTERFACE ItfBase EXTENDS __System.IQueryInterface
METHOD mbase : BOOL
END_METHOD
INTERFACE ItfDerived1 EXTENDS ItfBase
METHOD mderived1 : BOOL
END_METHOD
INTERFACE ItfDerived2 EXTENDS ItfBase
METHOD mderived2 : BOOL
END_METHOD
FUNCTION_BLOCK FB1 IMPLEMENTS ItfDerived1
METHOD mbase : BOOL
    mbase := TRUE;
END_METHOD
METHOD mderived1 : BOOL
    mderived1 := TRUE;
```

```
END_METHOD
END_FUNCTION_BLOCK
FUNCTION_BLOCK FB2 IMPLEMENTS ItfDerived2
METHOD mbase : BOOL
    mbase := FALSE;
END_METHOD
METHOD mderived2 : BOOL
    mderived2 := TRUE;
END_METHOD
END_FUNCTION_BLOCK
PROGRAM POU
VAR
    inst1 : FB1;
    inst2 : FB2;
    itfbase1 : ItfBase := inst1;
    itfbase2 : ItfBase := inst2;
    itfderived1 : ItfDerived1 := 0;
    itfderived2 : ItfDerived2 := 0;
    bTest1, bTest2, xResult1, xResult2: BOOL;
END_VAR
xResult1 := __QUERYINTERFACE(itfbase1, itfderived1); //
xResult = TRUE, itfderived1 <> 0
                                        // references the
instance inst1
xResult2 := __QUERYINTERFACE(itfbase1, itfderived2); //
xResult = FALSE, itfderived2 = 0
xResult3 := __QUERYINTERFACE(itfbase2, itfderived1); //
xResult = FALSE, itfderived1 = 0
xResult4 := __QUERYINTERFACE(itfbase2, itfderived2); //
xResult = TRUE, itfderived2 <> 0
                                        // references the
instance inst2
```

__QUERYPOINTER

Definition

This operator is not specified by the IEC 61131-3 standard.

At runtime, `__QUERYPOINTER` is assigning an interface reference to an untyped pointer. The operator returns a result with type BOOL. TRUE implies, that the conversion has been successfully executed.

Syntax

__QUERYPOINTER (<ITF_Source>, < Pointer_Dest>

For the first operand, the operator requires an interface reference or a function block instance of the intended type and for the second operand an untyped pointer. After execution of `__QUERYPOINTER`, the *Pointer_Dest* holds the address of the reference to the intended interface. In this case, the conversion is successful and the result of the operator returns TRUE. In all other cases, the operator returns FALSE. *Pointer_Dest* is untyped and can be cast to any type. The programmer has to ensure the actual type. For example, the interface could provide a method returning a type code.

A precondition for an explicit conversion is that the *ITF_Source* is an extension of the interface *__System.IQueryInterface*. This interface is provided implicitly and needs no library.

Example

```
TYPE KindOfFB
```

```
    (FB1 := 1, FB2 := 2, UNKOWN := -1);
END_TYPE
INTERFACE Itf EXTENDS __System.IQueryInterface
METHOD KindOf : KindOfFB
END_METHOD
FUNCTION_BLOCK F_BLOCK_1 IMPLEMENTS ITF
METHOD KindOf : KindOfFB
    KindOf := KindOfFB.FB1;
END_METHOD
FUNCTION_BLOCK F_BLOCK_2 IMPLEMENTS ITF
METHOD KindOf : KindOfFB
    KindOf := KindOfFB.FB2;
END_METHOD
FUNCTION CAST_TO_ANY_FB : BOOL
VAR_INPUT
    itf_in : Itf;
END_VAR
VAR_OUTPUT
    pfb_1: POINTER TO F_BLOCK_1 := 0;
    pfb_2: POINTER TO F_BLOCK_2 := 0;
END_VAR
VAR
    xResult1, xResult2 : BOOL;
END_VAR
IF itf_in <> 0
    CASE itf_in.KindOf OF
        KindOfFB.FB1:
            xResult1 := __QUERYPOINTER(itf_in, pfb_1);
        KindOfFB.FB2 THEN
            xResult2 := __QUERYPOINTER(itf_in, pfb_2);
    END_CASE
END_IF
CAST_TO_ANY_FB := xResult1 OR xResult2;
```

AND_THEN

Definition

This operator is not specified by the IEC 61131-3 standard. It is only allowed for programming in structured text (ST).

The AND_THEN performs an AND-operation of operands of type BOOL and BIT with short-circuiting mode. This has the following effect:

If all operands are TRUE, then the result of the operation is TRUE, otherwise it is FALSE.

When one operand is FALSE, then the expressions at the other operands are not evaluated (lazy evaluation). In this regard, the AND_THEN operator differs from the AND operator as defined in the standard IEC-61131-3. AND always evaluates all expressions, page 689.

In contrast to this, when using the standard IEC-operator AND, always all operands are evaluated, page 689.

OR_ELSE

Definition

This operator is not specified by the IEC 61131-3 standard. It is only allowed for programming in structured text (ST).

The OR_ELSE performs an OR-operation of operands of type BOOL and BIT with short-circuiting mode. This has the following effect:

If at least one of the operands is TRUE, then the result of the operation is TRUE, otherwise it is FALSE.

When one operand is TRUE, then the expressions at the other operands are not evaluated (lazy evaluation). In this regard, the OR_ELSE operator differs from the OR operator as defined in the standard IEC-61131-3. OR always evaluates all expressions, page 689.

Example

```
 VAR
  bEver: BOOL;
     bX1: BOOL;
  dw: DWORD := 16#000000FF;
END_VAR
bEver := FALSE;
bX := dw.8 OR_ELSE dw.1 OR_ELSE dw.1 OR_ELSE (bEver :=
TRUE);
```

dw.8 is FALSE and dw.1 is TRUE, thus the result of the operation (bX) is TRUE. However, the expression at the third input is not executed, bEver remains FALSE. If you used the standard OR operation instead, page 689, bEver were set to TRUE.

__TRY, __CATCH, __FINALLY, __ENDTRY

Definition

These operators are not specified by the IEC 61131-3 standard.

The __TRY, __CATCH, __FINALLY, __ENDTRY operators are used for specific handling of exceptions in IEC code. With these operators, you can execute specific statements in case an error is detected. Furthermore, the program is not stopped (as usual) when an exception is detected.

Syntax

```
__TRY
<statements_try>
__CATCH(exec)
<statements_catch>
__FINALLY
<statements_finally>
__ENDTRY
<statements_next>
```

Function

If an exception is detected while <statements_try> is executed (even in functions called from there), then <statements_catch> is executed. In that way, execution does not stop as it is the case when other exceptions are detected. However, there is a log message in the runtime system with information about the range offset and the type of exception that has been detected.

After executing <statements_catch>, <statements_finally> is automatically executed (if programmed), and then <statements_next> is executed.

The variable <exception> must be of type __SystemExceptionCode.

Example

If the statement in __TRY produces an exception, then program execution is not stopped, and then the statements in __CATCH are executed. Therefore, the function exc is executed. Then the statement in __ENDTRY is executed.

```
FUNCTION Tester : UDINT
VAR_INPUT
    count : UDINT;
END_VAR
VAR_OUTPUT
    strExceptionText : STRING;
END_VAR
VAR
    exc : __SYSTEM.ExceptionCode;
END_VAR
__TRY
Tester := tryFun(count := count, testcase := g_testcase);
//This statement is tested. If it produces an exception,
then the statement in __CATCH is executed first, and then
the statement in __FINALLY.
__CATCH(exc)
HandleException(exc, strExceptionText => strExceptionText);
__FINALLY
GVL.g_count := GVL.g_count + 2;
__ENDTRY
```

Use the **Stop execution on handled exceptions** command (see Menu Commands, Online Help) to stop execution at the error location despite the programmed exception handling.

Type __System.Exception

```
TYPE ExceptionCode:
( RTSEXCPT_UNKNOWN                                  :=
16#FFFFFFFF,
RTSEXCPT_NOEXCEPTION                                :=
16#00000000,
RTSEXCPT_WATCHDOG                                   :=
16#00000010,
RTSEXCPT_HARDWAREWATCHDOG                           :=
16#00000011,
RTSEXCPT_IO_CONFIG_ERROR                            :=
16#00000012,
RTSEXCPT_PROGRAMCHECKSUM                            :=
16#00000013,
RTSEXCPT_FIELDBUS_ERROR                             :=
16#00000014,
RTSEXCPT_IOUPDATE_ERROR                             :=
16#00000015,
RTSEXCPT_CYCLE_TIME_EXCEED                          :=
16#00000016,
RTSEXCPT_ONLCHANGE_PROGRAM_EXCEEDED                 :=
16#00000017,
RTSEXCPT_UNRESOLVED_EXTREFS                         :=
16#00000018,
RTSEXCPT_DOWNLOAD_REJECTED                          :=
16#00000019,
RTSEXCPT_BOOTPROJECT_REJECTED_DUE_RETAIN_ERROR      :=
16#0000001A,
RTSEXCPT_LOADBOOTPROJECT_FAILED                     :=
16#0000001B,
RTSEXCPT_OUT_OF_MEMORY                              :=
16#0000001C,
RTSEXCPT_RETAIN_MEMORY_ERROR                        :=
16#0000001D,
```

```
RTSEXCPT_BOOTPROJECT_CRASH                    :=
16#0000001E,
RTSEXCPT_BOOTPROJECTTARGETMISMATCH             :=
16#00000021,
RTSEXCPT_SCHEDULEERROR                        :=
16#00000022,
RTSEXCPT_FILE_CHECKSUM_ERR                    :=
16#00000023,
RTSEXCPT_RETAIN_IDENTITY_MISMATCH             :=
16#00000024,
RTSEXCPT_IEC_TASK_CONFIG_ERROR                :=
16#00000025,
RTSEXCPT_APP_TARGET_MISMATCH                  :=
16#00000026,
RTSEXCPT_ILLEGAL_INSTRUCTION                  :=
16#00000050,
RTSEXCPT_ACCESS_VIOLATION                     :=
16#00000051,
RTSEXCPT_PRIV_INSTRUCTION                     :=
16#00000052,
RTSEXCPT_IN_PAGE_ERROR                        :=
16#00000053,
RTSEXCPT_STACK_OVERFLOW                       :=
16#00000054,
RTSEXCPT_INVALID_DISPOSITION                  :=
16#00000055,
RTSEXCPT_INVALID_HANDLE                       :=
16#00000056,
RTSEXCPT_GUARD_PAGE                           :=
16#00000057,
RTSEXCPT_DOUBLE_FAULT                         :=
16#00000058,
RTSEXCPT_INVALID_OPCODE                       :=
16#00000059,
RTSEXCPT_MISALIGNMENT                         :=
16#00000100,
RTSEXCPT_ARRAYBOUNDS                          :=
16#00000101,
RTSEXCPT_DIVIDEBYZERO                         :=
16#00000102,
RTSEXCPT_OVERFLOW                             :=
16#00000103,
RTSEXCPT_NONCONTINUABLE                       :=
16#00000104,
RTSEXCPT_PROCESSORLOAD_WATCHDOG               :=
16#00000105,
RTSEXCPT_FPU_ERROR                            :=
16#00000150,
RTSEXCPT_FPU_DENORMAL_OPERAND                 :=
16#00000151,
RTSEXCPT_FPU_DIVIDEBYZERO                     :=
16#00000152,
RTSEXCPT_FPU_INEXACT_RESULT                   :=
16#00000153,
RTSEXCPT_FPU_INVALID_OPERATION                :=
16#00000154,
RTSEXCPT_FPU_OVERFLOW                         :=
16#00000155,
RTSEXCPT_FPU_STACK_CHECK                      :=
16#00000156,
RTSEXCPT_FPU_UNDERFLOW                        :=
16#00000157,
RTSEXCPT_VENDOR_EXCEPTION_BASE                :=
16#00002000
RTSEXCPT_USER_EXCEPTION_BASE                  :
=16#00010000 )
UDINT ; END_TYPE
```

__VARINFO

Definition

This operator is not specified by the IEC 61131-3 standard.

Syntax in the Declaration

```
<name of the info variable> : __SYSTEM.VAR_INFO; // Data
structure for info variable
```

Syntax for the Call

```
<name of the info variable> := __VARINFO( <variable name> );
// Call of the operator
```

Function

The __*VARINFO* operator provides information about a variable of the project at runtime. The information is stored as a data structure in a variable of data type __*SYSTEM.VAR_INFO*.

Example

At runtime, the variable *MyVarInfo* contains the information about the variable *MyVar*.

```
//Declaration
VAR
   MyVarInfo: __SYSTEM.VAR_INFO
   MyVAR: INT;
END_VAR

//Program code
MyVarInfo:= __VARINFO (MyVar);
```

Type *SYSTEM.VAR_INFO*

A variable with data type __*SYSTEM.VAR_INFO* contains the following elements:

Element	Data type	Initialization	Description
ByteAddress	DWORD	0	Address of the variable. **NOTE:** For bit access of a variable `<variable name>.<bit index>`, the address of the variable that contains the bit is given.
ByteOffset	DWORD	0	Offset of the variable address (in bytes).
Area	DINT	0	Number of the memory area (in the runtime system).
BitNr	INT	0	Number of bits (in bytes). **NOTE:** If the variable is not an integer data type, then `BitNr = -1`.
BitSize	INT	0	Size of the variable (in bits).
BitAddress	UDINT	0	Bit address of the variable.

Element	Data type	Initialization	Description
			Only valid if the variable is located in a %I, %Q, or %M area.
TypeClass	*TYPE_CLASS*	*TYPE_BOOL*	Data type class of the variable. Example: TYPE_INT, TYPE_ARRAY **NOTE:** For user-defined data types or function block instances, *TYPE_USERDEF* is delivered as the data type class.
TypeName	*STRING[79]*	''	Data type of the variable.
NumElements	UDINT	0	For arrays: Number of array elements.
BaseTypeClass	*TYPE_CLASS*	*TYPE_BOOL*	For arrays: Data type class of the elements in the array.
ElemBitSize	UDINT	0	For arrays: Bit size of an array element.
MemoryArea	*MEMORY_AREA*	*MEM_MEMORY*	Information about the memory area: • *MEM_GLOBAL*: Global memory area • *MEM_LOCAL*: Local memory area • *MEM_MEMORY*: %M • *MEM_INPUT*: %I • *MEM_OUTPUT*: %Q • *MEM_RETAIN*: Retain memory area
Symbol	*STRING[39]*	''	Variable name.
Comment	*STRING[79]*	''	Comment of the variable declaration.

'__POSITION'

Definition

This operator is not specified by the IEC 61131-3 standard.

At runtime, the '__POSITION' operator retrieves the position of a variable in the declaration part or in the implementation part of a POU. The operator has to be assigned variables of type STRING in the declaration part of in the implementation part.

Result of '__POSITION':

- Declaration part: 'Line <line number> (Decl)'
- Implementation part: 'Line <line number>, Column <column number> (Impl)'

Example

```
PROGRAM PROG1
VAR
        strPOS : STRING := __POSITION();  //Yields the line
number of this declaration
        strlocalPOS : STRING;
END_VAR
        strlocalPOS := __POSITION();  //Yields the line and
column number of this assignment
```

'__POUNAME'

Definition

This operator is not specified by the IEC 61131-3 standard.

At runtime, the '__POUNAME' operator retrieves the name of the POU that calls the operator '__POUNAME'. The result is of type STRING.

The result of the operator '__POUNAME' depends on the location where it is implemented:

Location of '__POUNAME'	Result
Program	Program name
Function	Function name
Function block	Function block name
Method	Method name qualified with the function block name
Get/Set accessor of a property	The property name + Get/Set qualified with the function block name
GVL	GVL name
Structure	Structure name
Data structure UNION	Union name

Example

```
PROGRAM PROG1
VAR
        strPOU : STRING := __POUNAME();  //Yields 'PROG1'
        strlocalPOU : STRING;
END_VAR

        strlocalPOU := __POUNAME();     //Yields 'PROG1'
```

__POOL

Definition

This operator is not specified by the IEC 61131-3 standard.

The __POOL operator is used to reference objects which are managed in the global POUs pool. It directly accesses objects in the **Global** node of the **Applications tree**.

Example

```
PROGRAM PLC_PRG
VAR
        svar_pou : STRING;
END_VAR

svar_pou := __POOL.POU();
```

Scope Operators

Definition

In extension to the IEC operators, there are several possibilities to disambiguate the access to variables or modules if the variables or module name is used multiple times within the scope of a project.

The following scope operators can be used:

- global scope operator
- global variable list name
- enumeration name
- library namespace
- global node operator

Global Scope Operator

An instance path starting with dot (.) opens a global scope (namespace). Therefore, if there is a local variable with the same name <*varname*> as a global variable, then .<*varname*> refers to the global variable.

Global Variable List Name

You can use the name of a global variable list as a namespace for the variables enclosed in this list. Thus, it is possible to declare variables with identical names in different global variable lists and, by preceding the variable name by <*global variable list name*>., it is possible to access the desired one.

Syntax

<global variable list name>.<variable>

Example

The global variable lists *globlist1* and *globlist2* each contain a variable named *varx*. In the following line, *varx* out of *globlist2* is copied to *varx* in *globlist1*:

```
globlist1.varx := globlist2.varx;
```

If a variable name declared in more than one global variable lists is referenced without the global variable list name as a preceding operator, a message will be generated.

Library Namespace

You can add the library namespace to a POU as a prefix separated by a dot to make the access to the POU unique. By default, the *namespace* of a library is identical to the library name.

Example: `LIB_A.FB_A`

Syntax

```
<library namespace>.<library POU>
```

Example

If a library which is included in a project contains the POU *FB_A* and there is also a POU *FB_A* defined locally in the project, then you can assign the name *LIB_A.FB_A* to the function block of the library in order to differentiate from the POU.

```
var1 := FB_A(in := 12); // Call of the project function
block FB_A
var2 := LIB_A.FB_A(in := 22); // Call of the library
function block FB_A
```

You can define another name for the namespace either in the **Project Information** when creating a library project in the **Project Information** (by default in the **Project** menu), or later in the **Properties** dialog box of an included library in the **Library Manager**.

Operator (#) for Direct Access

Local components of a function block can override library POUs with the same name, causing ambiguity and inhibiting access with the usual namespace convention. If this occurs, you can use the `#` operator to force direct access.

Syntax

```
<namespace> # <library POU path>
```

Example

```
TYPE Struct1 :
STRUCT
    Standard : Struct2;
END_STRUCT
END_TYPE


TYPE Struct2 :
STRUCT
    Concat : DINT;
END_STRUCT
END_TYPE


VAR_GLOBAL
    Standard : Struct2;
END_VAR


PROGRAM PLC_PRG
str := Standard.CONCAT('test1', '3444'); // using the "."
will generate an error, as there's a conflict btw.
"Standard" variable an library namespace.
str := Standard#CONCAT('test1', '3444'); // using '#" will
solve the ambiguity by referencing "CONCAT" function of
"Standard" library
```

Enumeration Name

You can use the type name of an enumeration to disambiguate the access to an enumeration constant. Therefore, it is possible to use the same constant in different enumerations.

The enumeration name has to precede the constant name, separated by a dot (.).

Syntax

<enumeration name>.<constant name>

Example

The constant *Blue* is a component of enumeration *Colors* as well as of enumeration *Feelings*.

```
color := Colors.Blue; // Access to enum value Blue in type
Colors
feeling := Feelings.Blue; // Access to enum value Blue in
type Feelings
```

Global Node Operator

Variables declared in GVLs or POUs of the **Global** node of the **Applications tree** can be accessed by prefixing them with the operator `"__POOL."`.

Initialization Operator

INI Operator

Overview

You can use the `INI` operator to initialize retain variables which are provided by a function block instance used in the POU.

Assign the operator to a boolean variable.

> **NOTE:** The `INI` operator is obsolete. The method `FB_init` replaces the `INI` operator. For further information about the `FB_init` method, refer to the chapter FB_init, FB_reinit Methods, page 582. However, the operator can still be used for keeping compatibility with projects imported from earlier EcoStruxure Machine Expert versions.

Syntax

<bool-variable> := INI(<FB-instance, TRUE|FALSE)

If the second parameter of the operator is set to TRUE, all retain variables defined in the function block *FB* will be initialized.

Example in ST

fbinst is the instance of function block *fb*, in which a retain variable *retvar* is defined.

Declaration in POU

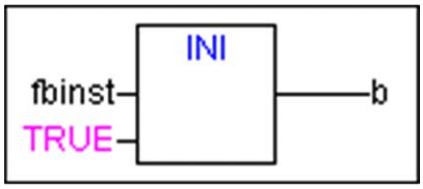
```
fbinst:fb;
b:bool;
```

Implementation part

```
b := INI(fbinst, TRUE);
```

```
ivar:=fbinst.retvar (* => retvar gets initialized *)
```

Example of Operator Call in FBD



Multicore Operators

Overview

Multicore operators are not specified by the standard IEC 61131-3.

They are used for task synchronization, especially when working on multicore platforms. The operators *TEST_AND_SET* and *__COMPARE_AND_SWAP* are used for similar tasks.

__CURRENTTASK

Overview

This operator is not specified by the standard IEC 61131-3.

In runtime mode, the operator provides information about the IEC task that is running.

This function is not available for all controllers. Consult the *Programming Guide* specific to your controller for further information.

The `__CURRENTTASK` operator allows for access to a structure with two variables:

- *TaskIndex*: Zero-based index that identifies the task.
- *pTaskInfo*: Detailed information about the running task. It can be assigned to a `POINTER TO Task_Info2` from the library CmpIecTask.

The operator cannot be used in the declaration of a POU. This would result in a detected compiler error. If the running task cannot be determined, then the *TaskIndex -1* and the *pTaskInfo* are zero.

Example

```
//Declaration
VAR
    idx : INT;
    pInfo : POINTER TO Task_Info2;
END_VAR

//Program code

idx := __CURRENTTASK.TaskIndex;
pInfo := __CURRENTTASK.pTaskInfo;
```

__COMPARE_AND_SWAP

Overview

This operator is not specified by the standard IEC 61131-3.

It can be used for implementing a semaphore, for example, to help ensure exclusive access to a variable that is written by different tasks.

The following elements are assigned to `__COMPARE_AND_SWAP`:

- Pointer to a data type *__XWORD* variable
- Old value as input, can also be data type *__XWORD* variable
- New value as input, can also be data type *__XWORD* variable

Example: `bMutex := __COMPARE_AND_SWAP(ADR(dwSynch), dwOld, dwNew);`

The referenced *__XWORD* variable is compared with the old value. If both are equal, the new value is written. The result of the function is TRUE if the new value is written.

> **NOTE:** The data type __XWORD is replaced by DWORD on 32-bit systems and by LWORD on 64-bit systems by the compiler.

This atomic operation cannot be interrupted by another task, even on multicore platforms.

Example

The following example illustrates a typical use case: Implementing exclusive access to a variable of type STRING which is addressed via the *pstrOutput* pointer.

As the access to a variable of type STRING is not atomic, the contents may be inconsistent if multiple tasks write to the same string at the same time.

Using the `__COMPARE_AND_SWAP` operator function allows you to write the same STRING variable in different tasks.

```
FUNCTION ExclusiveStringWrite : BOOL
VAR_INPUT
    strToWrite : STRING;
    pstrOutput : POINTER TO STRING;
END_VAR
VAR_STAT
  dwSynch : __XWORD;
END_VAR
VAR
    bMutex: BOOL;
END_VAR

bMutex:= __COMPARE_AND_SWAP(ADR(dwSynch), 0, 1);
(* compare with 0 and write 1 as atomic operation *)
IF bMutex THEN                        // bMutex is TRUE if
write could be done
    pstrOutput^ := strToWrite;      // Now you can write
safely on the string
    dwSynch := 0;                   // The __XWORD variable
must be reset.
    ExclusiveStringWrite := TRUE; // Writing was successful
ELSE
    ExclusiveStringWrite := FALSE; // Writing was not
successful
END_IF
```

__XADD

Overview

This operator is not specified by the standard IEC 61131-3.

It can be used for implementing an atomic counter. Simple addition operations executed on integer variables, for example `iTest := iTest + 1;`, are not executed atomically. This implies the risk that another process may access the variable between the read and write operation.

If the counter is incremented in multiple tasks, the result of the counter can be less than the number of counting cycles.

Example: The initial value of the variable is `0`. Two tasks execute the above code one time each. After that, the variable can have the value `1`. This may cause issues if arrays are being processed in multiple tasks, but nevertheless each processing cycle requires a unique index for the array.

When the `__XADD` operator is called, the following procedure applies:

1. `__XADD` retrieves a pointer to a type `DINT` variable as the first summand.
2. `__XADD` retrieves a type `DINT` value as the second summand.
3. `__XADD` returns the old value of the first summand and in the same step adds the second summand to the first summand.

Function call example: `diOld := __XADD(ADR(diVar), deAdd);`

Example

The following example illustrates a typical use case: An array is to be populated from two tasks in such a way that all positions in the array are used and no position is overwritten.

Using the `__XADD` operator function allows multiple tasks to populate a Boolean array.

```
FUNCTION WriteToNextArrayPosition : BOOL
VAR_EXTERNAL
    g_diIndex : DINT;  //                Index and array are
globally defined and used by multiple tasks
    g_boolArray : ARRAY [0..1000] OF BOOL;
END_VAR
VAR_INPUT
    bToWrite : BOOL;
END_VAR
VAR
    diIndex : DWORD;
END_VAR

diIndex := __XADD(ADR(g_diIndex), 1);   // Returns a unique
index
WriteToNextArrayPosition := FALSE;
IF (diIndex >= 0 AND diIndex <= 1000) THEN
    g_boolArray[diIndex] := bToWrite;  //Writes to unique
index
    WriteToNextArrayPosition := TRUE;  // TRUE: Array was
not full yet
END_IF
```

TEST_AND_SET

Overview

TEST_AND_SET can be used for implementing a semaphore, for example, to help ensure exclusive access to a variable that is written by different tasks.

NOTE: TEST_AND_SET is not supported by Modicon M262 Logic/Motion Controllers. Further, it is not specified by the standard IEC 61131-3.

The following procedure is applied to TEST_AND_SET:

1. A variable of type DWORD is assigned as input. Write access to this variable must be possible.
2. The variable is set to 1.
3. The previous value is returned as the result.

This operation cannot be interrupted by another task; this also applies to multicore platforms.

Example: The call in the program is dwOldValue := TEST_AND_SET(dw);. The variables *dwOldValue* and *dw* must be of data type DWORD.

Example

The following example illustrates a typical use case: Implementing exclusive access to a variable of type STRING which is addressed via the *pstrOutput* pointer.

As the access to a variable of type STRING is not atomic, the contents may be inconsistent if multiple tasks write to the same string at the same time.

Using the TEST_AND_SET operator function allows you to write the same STRING variable in different tasks.

```
FUNCTION ExclusiveStringWrite : BOOL
VAR_INPUT
    strToWrite : STRING;
    pstrOutput : POINTER TO STRING;
END_VAR
VAR_STAT
    dwSynch : DWORD;
END_VAR
VAR
    dwOldValue: DWORD;
END_VAR

dwOldValue := TEST_AND_SET(dwSynch);  // Write the 1 and
read the old value at the same time
IF dwOldValue = 0 THEN                // 0 means: no other
task is currently writing
   pstrOutput^ := strToWrite;         // Now you can write
on the string
   dwSynch := 0;                      // The DWORD must be
reset
   ExclusiveStringWrite := TRUE;      // Writing was
successful
ELSE
   ExclusiveStringWrite := FALSE;     // Writing was not
successful
```

Operands

What's in This Chapter

Constants 746
Variables 754
Addresses 760
Functions 761

Constants

BOOL Constants

Overview

BOOL constants are the logical values TRUE and FALSE.

Refer to the description of the data type BOOL, page 635.

TIME/LTIME Constants

Overview

TIME constants are used to operate the standard timer modules. The time constant TIME is of size 32 bit, resolution in milliseconds as specified by the IEC 61131-3 standard. Additionally, LTIME is supported as an extension to the standard as time base for high-resolution timers. LTIME is of size 64 bit and provides a resolution in nanoseconds.

The library standard64.lib provides functions for WSTRING strings.

Syntax for TIME Constant

t#<time declaration>

Instead of `t#`, you can also use the following:

- `T#`
- `time#`
- `TIME#`

The time declaration can include the following time units. They have to be used in the following sequence, but it is not required to use all of them.

- `D | d`: days
- `H | h`: hours
- `M | m`: minutes
- `S | s`: seconds
- `MS | ms`: milliseconds

Examples of correct TIME constants in an ST assignment

Example	Description
`TIME1 := T#14ms;`	–
`TIME1 := T#100S12ms;`	(* The highest component may be allowed to exceed its limit *)
`TIME1 := t#12h34m15s;`	–

Examples of incorrect usage

Example	Description
`TIME1 := t#5m68s;`	(* limit exceeded in a lower component *)
`TIME1 := 15ms;`	(* T# is missing *)
`TIME1 := t#4ms13d;`	(* incorrect order of entries *)

Syntax for LTIME Constant

LTIME#<time declaration>

The time declaration can include the time units as used with the TIME constant and additionally:

- us: microseconds
- ns: nanoseconds

Examples of correct LTIME constants in an ST assignment:

```
LTIME1 := LTIME#1000d15h23m12s34ms2us44ns
LTIME1 := LTIME#3445343m3424732874823ns
```

For further information, refer to the description of the Date and Time Data Types, page 639.

DATE Constants

Overview

Use these constants to enter dates.

Syntax

d#<date declaration>

Instead of `d#` you can also use the following:

- `D#`
- `date#`
- `DATE#`

Enter the date declaration in format `<date keyword>#<year>-<month>-<day>`.

DATE values are internally handled as DWORD values, containing the time span in seconds since 01.01.1970, 00:00 clock, with an upper limit of DATE#2106-2-7.

Examples

```
DATE#1996-05-06
d#1972-03-29
```

For further information, refer to the description of the Date and Time Data Types, page 639.

LDATE Constants

Overview

Use the LDATE constant to specify a date.

Syntax

ld#<date declaration>

Instead of `ld#` you can also use the following:

- `LD#`
- `ldate#`
- `LDATE#`

Enter the date declaration in format `<date keyword>#<year>-<month>-<day>`.

LDATE values are 64-bit types and contain the time span in nanoseconds since 01.01.1970, 00:00 clock until 21.07.2554.

Examples
```
LDATE#2018-8-8
LD#1996-05-06
```

For further information, refer to the description of the Date and Time Data Types, page 639.

DATE_AND_TIME Constants

Overview

DATE constants and TIME constants can also be combined to form so-called DATE_AND_TIME constants.

Syntax

`dt#<date and time declaration>`

Instead of `dt#` you can use the following:

- `DT#`
- `date_and_time#`
- `DATE_AND_TIME#`

Enter the date and time declaration in format <year-month-day-hour:minute:second>.

You can enter seconds as real numbers. This allows you to specify fractions of a second.

DATE_AND_TIME values are internally handled as DWORD values, containing the time span in seconds since 01.01.1970, 00:00 clock until February 7, 2106 06:28:15.

Examples
```
DATE_AND_TIME#1996-05-06-15:36:30
dt#1972-03-29-00:00:00
```

For further information, refer to the description of the Date and Time Data Types, page 639.

LDATE_AND_TIME Constants

Overview

Use the LDATE_AND_TIME (LDT) constant to specify a 64-bit date and time.

Syntax

`ldt#<date and time declaration>`

Instead of `ldt#` you can use the following:

- `LDT#`
- `ldate_and_time#`
- `LDATE_AND_TIME#`

Enter the date and time declaration in format <year-month-day-hour:minute:second>.

You can enter seconds as real numbers. This allows you to specify fractions of a second.

LDATE_AND_TIME values are 64-bit types and contain the time span in nanoseconds since 01.01.1970, 00:00 clock until 21.07.2554, 23:34:33.709551615.

Examples
```
LDATE_AND_TIME#1996-05-06-15:36:30
ldt#1972-03-29-00:00:00
LDT#2018-08-08-13:33:20.5
```

For further information, refer to the description of the Date and Time Data Types, page 639.

TIME_OF_DAY Constants

Overview

Use this type of constant to store times of the day.

Syntax

tod#<time declaration>

Instead of `tod#` you can also use the following:

- `TOD#`
- `time_of_day#`
- `TIME_OF_DAY#`

Enter the time declaration in format `<hour:minute:second>`.

You can enter seconds as real numbers. This allows you to specify fractions of a second.

TIME_OF_DAY values are internally handled as DWORD values, containing the time span in milliseconds since 00:00 clock.

Examples
```
TIME_OF_DAY#15:36:30.123
tod#00:00:00
```

For further information, refer to the description of the Date and Time Data Types, page 639.

LTIME_OF_DAY Constants

Overview

Use this type of constant to store times of the day in 64 bit format.

Syntax

ltod#<time declaration>

Instead of `ltod#` you can also use the following:

- `LTOD#`
- `ltime_of_day#`
- `LTIME_OF_DAY#`

Enter the time declaration in format `<hour:minute:second>`.

You can enter seconds as real numbers. This allows you to specify fractions of a second.

LTIME_OF_DAY values are internally handled as LWORD values, containing the time span in nanoseconds since 00:00 clock.

Examples

```
LTIME_OF_DAY#15:36:30.123456789
ltod#00:00:00
```

For further information, refer to the description of the Date and Time Data Types, page 639.

Number Constants

Overview

Number values can appear as binary numbers, octal numbers, decimal numbers, and hexadecimal numbers. Integer values that are not decimal numbers are represented by the base followed by the number sign (#) in front of the integer constant. The values for the numbers 10...15 in hexadecimal numbers are represented by the letters A...F.

You can include the underscore character within the number.

Examples

`14`	(decimal number)
`2#1001_0011`	(dual number)
`8#67`	(octal number)
`16#A`	(hexadecimal number)

These number values can be of type:

- BYTE
- WORD
- DWORD
- SINT
- USINT
- INT
- UINT
- DINT

- UDINT
- REAL
- LREAL

Implicit conversions from larger to smaller variable types are not permitted. Therefore, a DINT variable cannot simply be used as an INT variable. Use the type conversion functions, page 706.

REAL/LREAL Constants

Overview

REAL and LREAL constants can be given as decimal fractions and represented exponentially. Use the decimal point as decimal separator according to the International System of Units (SI).

Examples

REAL/LREAL Constant	Description
`7.4`	Do not use `7,4` because the comma will result in a compiler error.
`1/3.0`	Decimal fraction for 0.333333343. **NOTE:** In the case of division of integer types, the result remains an integer type. The value is rounded. For example, 1/3 yields 0 as the result.
`1.64e+009`	Do not use `1,64e+009` for exponential notations because the comma will result in a compiler error.

For limits of the REAL and LREAL data types, refer to the description of the *Standard Data Types*, page 637.

String Constants

Overview

A string constant is an arbitrary sequence of characters. STRING, page 638 constants are preceded and followed by single quotation marks. WSTRING, page 639 constants are preceded and followed by double quotation marks. The characters are coded according to the character set specified in ISO/IEC 8859-1. You may also enter blank spaces and special characters (special characters for different languages, like accents or umlauts).

If the **UTF8 Encoding for STRING** option is enabled in the **Project Settings > Compile options** dialog box, the string literal is interpreted in UTF-8 format (for further information, refer to the *Project Settings - Compile Options* chapter in the Menu Commands Online Help).

NOTE: A compatibility check is not performed for string literals, so you can enter all characters in the text editor. The compiler, however, compiles unknown characters as `?`.

Hexadecimal Code

In strings, the combination of the dollar sign ($) followed by 2 hexadecimal numbers is interpreted as a hexadecimal code according to the coding in ISO/IEC 8859-1. The code corresponds to ASCII code. In addition, note the special cases presented in the table.

Combinations of characters starting with a dollar sign which are interpreted as hexadecimal code:

String with $ code	Interpretation
'$<8-bit code>'	8-bit code: Two-digit hexadecimal number that is interpreted according to ISO/IEC 8859-1.
'$41'	A
'$9A'	©
'$40'	@
'$0D'	Control character: Line break (corresponds to '$R')
'$0A'	Control character: New line (corresponds to '$L' and '$N')

Special Cases of a STRING

Combinations of characters starting with a dollar sign which have a specific meaning:

String with $ code	Interpretation
'$L', '$l'	Control character: Line feed (corresponds to '$0A')
'$N', '$n'	Control character: New line (corresponds to '$0A')
'$P' '$p'	Control character: Form feed
'$R', '$r'	Control character: Line break (corresponds to '$0D')
'$T', '$t'	Control character: Tab
'$$'	Dollar sign $
'$''	Single straight quotation mark: '

Examples

Constant declaration of a STRING:

```
VAR CONSTANT
    constA : STRING := 'Hello world';
    constB : STRING := 'Hello world $21'; // Hello world!
END_VAR
```

Examples of WSTRING declarations:

```
wstr:WSTRING:="This is a WString";
wstr10 : WSTRING(10) := "1234567890";
```

Constant: UTF8# String

UTF-8-encoded string literals are supported as a single-byte STRING.

Syntax
```
UTF8#' <string literal> '
```

Constant declaration
```
VAR CONSTANT
    constA : STRING := UTF8#'aäoöuü';
    constB : STRING := UTF8#'Hello Allgäu $21'; // Hello
Allgäu!
END_VAR
```

NOTE: To help ensure correct monitoring, add the attribute `'monitoring_encoding' := 'UTF-8'`.

For further information, refer to Attribute monitoring_encoding, page 618.

Also refer to

- Project-Wide UTF-8 Encoding, page 643
- Encoding Single Literals in UTF-8 Format, page 644

Character Constant UCHAR

Overview

The character literal for a single Unicode character has the base type UDINT. The type prefix UCHAR# is prefixed to the literal as an identifier. The numerical value of the character literal corresponds to the code point of the Unicode standard.

Syntax

```
UCHAR#' <single character> '
```

Literal

```
udiChar : UDINT := UCHAR#'Ω'; // cp 3603
```

Typed Constants / Typed Literals

Overview

Basically, in using IEC constants, the smallest possible data type will be used. An exception is REAL/LREAL constants where LREAL is always used. If another data type has to be used, use typed literals (typed constants) without the necessity of explicitly declaring the constants. For this purpose, the constant will be provided with a prefix which determines the type.

Syntax

```
<Type>#<Literal>
```

<Type> is the desired data type. Possible entries are:

- BOOL
- SINT
- USINT

- BYTE
- INT
- UINT
- WORD
- DINT
- UDINT
- DWORD
- REAL
- LREAL

Write the type in uppercase letters.

`<Literal>` specifies the constant. The data entered has to fit within the data type specified in `<Type>`.

Example
```
var1:=DINT#34;
```

If the constant cannot be converted to the target type without data loss, a message will be generated.

You can use typed literals wherever normal constants can be used.

Variables

Variables

Overview

You can declare variables either locally in the declaration part of a POU or in a global variable list (GVL) or in a persistent variables list or in the I/O mapping of devices.

Refer to the chapter *Variables Declaration*, page 550 for information on the declaration of a variable, including the rules concerning the variable identifier and multiple use.

It depends on the data type, page 635 where a variable can be used.

You can access available variables through the **Input Assistant**.

Accessing Variables for Arrays, Structures, and POUs

The table lists the respective syntax for accessing arrays, structures, and POUs:

Syntax	Access to
<array name>[Index1, Index2]	2-dimensional array, page 651 components
<structure name>.<variable name>	structure, page 661 variables
<function block name>|<program name>. <variable name>	function block and program variables

Addressing Bits in Variables

Overview

In integer variables, individual bits can be accessed. For this purpose, append the index of the bit to be addressed to the variable and separate it by a dot. You can give any constant to the bit index. Indexing is 0-based.

If your processor does not support direct bit access, you can use a semaphore (*SysSemEnter*) or a similar technique to help prevent competing bit access. However, to avoid access by two tasks, it is a good practice to execute the bit access within one task only.

Syntax

<variablename>.<bitindex>

Example

```
a : INT;
b : BOOL;
...
a.2 := b;
```

The third bit of the variable `a` will be set to the value of the variable `b`, therefore variable `a` will equal 3.

If the index is greater than the bit width of the variable, the following message will be generated:

'Index '<n>' outside the valid range for variable '<var>'!'

Bit addressing is possible with variables of the following data types:

- SINT
- INT
- DINT
- USINT
- UINT
- UDINT
- BYTE
- WORD
- DWORD

If the data type does not allow bit accessing, the following message will be generated:

'Invalid data type '<type>' for direct indexing'.

Do not assign bit access to a `VAR_IN_OUT` variable.

Bit Access Via a Global Constant

If you have declared a global constant defining the bit index, you can use this constant for a bit access.

Example for a bit access via a global constant and on a variable:

1. Declaration of the global constant in a global variable list

The variable `enable` defines the bit that is accessed:

```
VAR_GLOBAL CONSTANT
    enable:int:=2;
END_VAR
```

2. Bit access on an integer variable

Declaration in POU:

```
VAR
    xxx:int;
END_VAR
```

Bit access:

```
xxx.enable := true; (* -> the third bit in variable xxx will
be set TRUE *)
```

Bit Access on BIT Data Types

The BIT data type is a special data type which is only allowed in structures. You can then access each bit individually by using the component name. For further information, refer to Bit Access in Structures, page 663.

Example: Bit access on BIT data types

Declaration of structure

```
TYPE ControllerData :
STRUCT
    Status_OperationEnabled : BIT;
    Status_SwitchOnActive : BIT;
    Status_EnableOperation : BIT;
    Status_Error : BIT;
    Status_VoltageEnabled : BIT;
    Status_QuickStop : BIT;
    Status_SwitchOnLocked : BIT;
    Status_Warning : BIT;
END_STRUCT
END_TYPE
```

Declaration in POU

```
VAR
    ControllerDrive1:ControllerData;
END_VAR
```

Bit access

```
ControllerDrive1.Status_OperationEnabled := TRUE;
```

Symbolic Bit Access in Function Block Instances

In function blocks, you can declare variables for individual bits.

Example:

```
FUNCTION_BLOCK FB_Controller
VAR_INPUT
        bitSwitchOnActive : BIT;
        bitEnableOperation : BIT;
        bitVoltageEnabled : BIT;
        bitQuickStop : BIT;
        bitSwitchOnLocked : BIT;
END_VAR
```

```
VAR_OUTPUT
        bitOperationEnabled : BIT;
        bitError : BIT;
        bitWarning : BIT;
END_VAR
VAR
END_VAR
;

PROGRAM PLC_PRG
VAR
        fbController : FB_Controller;
END_VAR
// Symbolic bit access to bitSwitchOnActive
fbController(bitSwitchOnActive := TRUE);
```

Partial Variable Access

Overview

Conforming with the IEC 61131-3 standard, partial variable access is possible in the case of IEC variables of data type BYTE, WORD, DWORD or LWORD (data type ANY, ANY_<type>), page 640.

Syntax

```
<variable name>. % <partial type> <partial index>
```

A space is allowed after the dot operator (`.`). No spaces are allowed after the percent sign (`%`) and after the partial type.

Element	Description
`<variable name>`	Name of the `ANY_BIT` variable of the type BYTE, WORD, DWORD, or LWORD.
`<partial type>`	• `X` for BIT access • `B` for BYTE access • `W` for WORD access • `D` for DWORD access • `L` for LWORD access Access to __XWORD is also allowed and behaves like a DWORD or LWORD depending on the pointer size. Access to BOOL is not allowed even if the BOOL data type is part of the `ANY_BIT` types.
`<partial index>`	From 0 to the index maximum.

Usage

Partial access is allowed for non-temporary variables such as:

- User-defined variables
- Fields
- Array accesses
- Dereferencing

Partial access is **not allowed** for:

- Temporary results
- Literals

The expression itself is not temporary and can be used as an expression in `REFERENCE TO` or as an `ADR` operator.

Examples

```
// Implementation
VAR
    PartialVarB: INT;
    idx: INT;
    PartialVarW: WORD;
    tempVariable: DWORD;
    PartialVarD: DWORD;
    ptr: POINTER TO LWORD;
    variable: DWORD;
    PartialVarX: BOOL;
    array1 : ARRAY[1..10] OF BYTE;
END_VAR;

PartialVarB := GVL.Variable.%B0;
PartialVarX := array1[idx].%X0;
PartialVarW := tempVariable.%W1;
PartialVarD := ptr^.%D1;
PartialVarB := variable.%W1.%B1;
```

Partial access is **not allowed** in the case of function calls, index expressions, literals, or properties as indicated in the following **negative example**:

```
FunctionCall().%B0
(1+index).%B0
(DWORD#16#12345678).%B0
Property.%B0
```

Index Maximum of Partial Types

Partial access is possible only if the called expression has a conforming type and the partial index is less than or equal to the maximum partial index. Otherwise a compiler error is reported.

Partial type	Access type	Maximum of partial index
X		
	BYTE	7
	WORD	15
	DWORD	31
	LWORD	63
B		
	BYTE	0
	WORD	1
	DWORD	3
	LWORD	7
W		
	BYTE	Not supported
	WORD	0
	DWORD	1
	LWORD	3
D		
	BYTE	Not supported
	WORD	Not supported
	DWORD	0
	LWORD	1
L		
	BYTE	Not supported
	WORD	Not supported
	DWORD	Not supported
	LWORD	1

Examples for Addresses of Partial Types

You can create the address from a partial access expression of partial type `B`, `W`, `D`, and `L` as indicated in this example:

```
ADR(var%.B0)
referenceVariable REF= var.%W1
```

It is **not allowed** to create the address from a partial access expression of partial type `X` as indicated in the following **negative example**.

```
ADR(var.%X0)
ref REF= var.%X0
```

It is **not allowed** to assign a partial access expression of type BIT to a `VAR_IN_OUT` variable.

Addresses

Direct Addresses

Overview

A direct address specified in EcoStruxure Machine Expert contains the following information:

- Information on the memory location.
- Memory format (size)
- Offset of the memory location. The offset is specified by an integer number, which in case of a bit address is followed by a dot and a number for the position of the bit.

Syntax

%<memory area prefix><size prefix><number|.number|.number....>

The following memory area prefixes are supported:

`I`	input (physical inputs via input driver, sensors)
`Q`	output (physical outputs via output driver, actors)
`M`	memory location

The following size prefixes are supported:

`X`	single bit
`B`	byte (8 bits)
`W`	word (16 bits)
`D`	double word (32 bits)

Examples

Example address	Description
`%QX7.5`	output bit 7.5
`%Q7.5`	
`%IW215`	input word 215
`%QB7`	output byte 7
`%MD48`	double word in memory position 48 in the memory location
`ivar AT %IW0: WORD;`	example of a variable declaration including an address assignment For further information, refer to the `AT` Declaration chapter, page 562.

NOTE: The memory size for input, output, and memory data (declarations with `AT %I`, `%Q` and `%M`) is predefined by the target device and can be overwritten in the properties of an application object (see Menu Commands, Online Help) for PacDrive controllers (PacDrive LMC Eco, PacDrive LMC Pro/Pro2).

Byte Addressing Mode and Word Addressing Mode

Devices either use byte addressing mode or word addressing mode.

Examples

Mode	Example
Byte addressing	`ADR(%IW1) = ADR(%IB1)`
Word addressing	`ADR(%IW1) = ADR(%IB2)`

The range for the second element of the bit address that is the number following the dot, is as follows:

- byte addressing mode: 0...7
- word addressing mode: 0...15

Also for the handling of bit addresses, you can configure the devices differently. They are then interpreted correspondingly by the EcoStruxure Machine Expert compiler.

Example: In a byte-addressing device, the bit address `%IX2.5` addresses byte 2 (`IB2`). In a word-addressing device, however, it addresses word 2, which refers to a different location within the memory.

> **NOTE:** Boolean values are allocated bytewise if no explicit single-bit address is specified. Example: A change in the value of `varbool1 AT %QB7` affects the range from `QX0.0` to `QX0.7`.

Functions

Functions

Overview

In ST, a function call can be used as an operand.

Example

```
Result := Fct(7) + 3;
```

For a general description of functions and their declarations, refer to the *Function* description in the *Program Organization Unit (POU)* section of this Programming Guide, page 179.

TIME() Function

This function returns the time (based on milliseconds) which has been passed since the system was started.

The data type is TIME.

Example in IL

```
TIME
ST systime  (* Result for example: T#35m11s342ms *)
```

Example in ST

```
systime:=TIME();
```

Templates

What’s in This Part

General Information about Templates 763
Managing Device Templates 770
Managing Function Templates 779

General Information about Templates

What's in This Chapter

Templates 763

Templates

General Information About Templates

Overview

EcoStruxure Machine Expert provides templates in order to make dedicated control and visualization functionality that has been developed in one project easily available to other projects. They help to standardize the usage of field devices and application functions throughout different projects.

The following types of templates are available:

- Device templates that are associated with a single field device or I/O module
- Function templates that are associated with a high-level application function

Various templates are provided, but you can also create your own templates for functionalities you want to make available in other projects.

Creating Your Own Templates

Perform the following steps:

Step	Action
1	Create your functionality within a project and test it with the corresponding hardware or in the simulation.
2	Save the functionality in a template library.
3	Open another project and select the template from the template library in order to make the functionality available to this project.

General Notes

When using templates, note the following:

- Templates are not controller-specific and can therefore be made available for any controller. Verify that the controller to which you add the template is capable of executing the functionality contained in the template.
- After the template has been installed, you can freely adapt the created objects to your individual requirements.
- It is possible to install one template several times on the same controller device. In order to avoid naming conflicts when creating the objects, they are renamed automatically during installation. For further information, refer to the *Naming of Objects* section of the *Adding Devices from Template* chapter, page 772.
- Templates do not support the use of direct representations of variables (for example `%IX2.0`).

  However, you can use direct representations with an incomplete address specification (for example `%I*`). For further information, refer to the chapter *Variables configuration - VAR_CONFIG*, page 578.

NOTE: Although this form of placeholder for direct addresses is available, avoid direct addressing in your programs, and use symbolic addressing wherever and whenever possible.

Supported I/O Modules

Templates can include the following I/O modules:

- TM2
- TM3
- TM5

Supported Fieldbusses

Templates can include field devices that are linked to the following fieldbusses:

- CANopen
- Modbus serial line (Modbus IOScanner)
- Modbus TCP IO Scanner
- SoftMotion General Drive Pool
- CANmotion
- Ethernet IP Scanner
- Sercos3

Administration of Templates

Overview

The following paragraphs provide an overview of how to create new or change existing device or function templates and to save them as files for transferring them to other PCs.

Template Libraries

Template libraries contain the definition of several device or function templates.

Write Protection

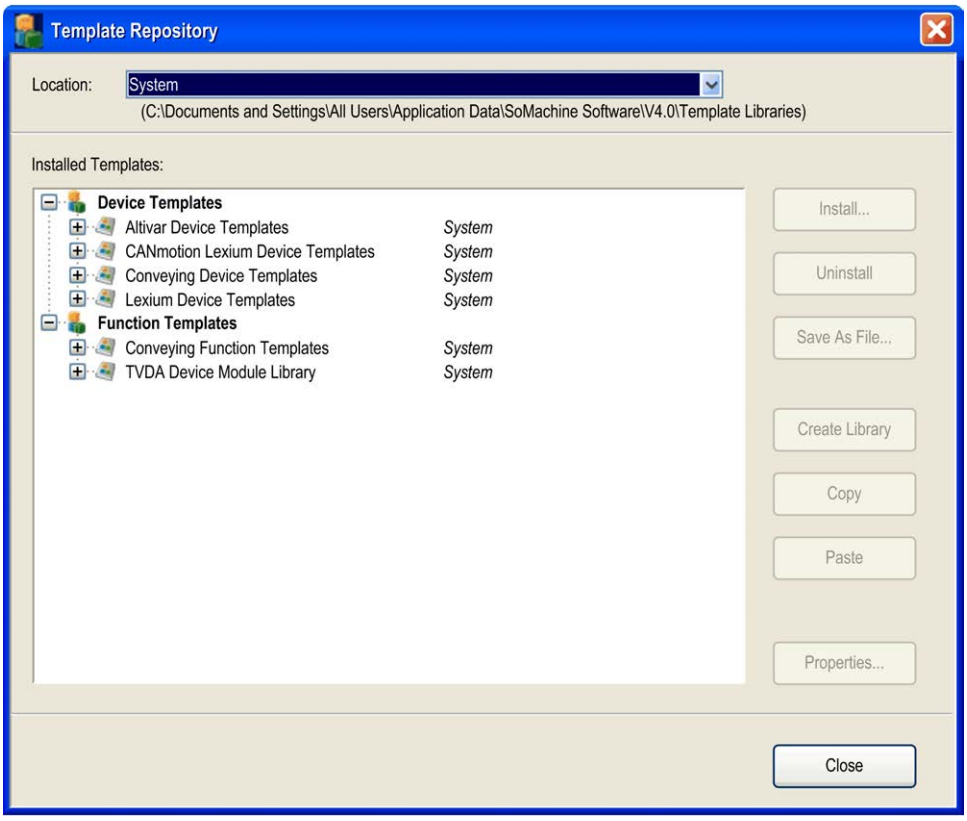
The standard template libraries included in the scope of delivery are write-protected, which means that they cannot be deleted or renamed.

NOTE: You cannot change write-protected libraries (uninstalling individual templates or changing names), but you can completely uninstall them.

Template Administration

For administration of the available device and function templates, select **Tools > Template Repository** in the Logic Builder.

The **Template Repository** dialog box opens:



From the **Location** list, select the type of templates to be displayed in the **Installed Templates** box:

- **<All locations>** is selected by default: all available device and function templates are displayed
- **Legacy** displays the device and function templates of EcoStruxure Machine Expert V3.1 (if installed)
- **User**: displays only those device and function templates that you have created or installed
- **System**: displays the standard device and function templates delivered by EcoStruxure Machine Expert

The path to the directory where the template libraries are stored is displayed below the **Location** field.

The **Installed Templates** box lists the installed templates in 2 groups: **Device Templates** and **Function Templates**. Each template library can either contain device templates or function templates.

Installing Additional Template Libraries

To add additional template libraries to this list, proceed as follows:

Step	Action
1	Click the **Install** button in the **Template Repository** dialog box. **Result**: A **File open** dialog box opens.
2	Browse to the folder where the template library file you want to install is saved.
3	Select the library file you want to install and click **OK**. **Result**: The selected template library is installed and is indicated in the **Template Repository** dialog box, including the device or function templates it contains.

Removing Template Libraries

To remove a template library, proceed as follows:

Step	Action
1	In the **Installed Templates** list of the **Template Repository** dialog box, select the template library you want to remove.
2	To remove the selected template library, click the **Uninstall** button. **Result**: The selected template library is removed from the installation.

Renaming Template Libraries

To rename a template library, proceed as follows:

Step	Action
1	In the **Installed Templates** list of the **Template Repository** dialog box, select the template library you want to rename.
2	Click the name of the template library you want to change. **Result**: A box opens.
3	Enter the new name in the box and press **Enter** or leave the box. **Result**: The template library is now assigned to the new name.

Creating a New Template Library

To create a new template library, proceed as follows:

Step	Action
1	To create a new template library, select the option **User** or **<All locations>** from the **Location** list.
2	To create a new template library for device templates, select the **Device Templates** node in the **Installed Templates** list and click the **Create Library** button. **Result**: A new template library with a default name is added at the bottom of the **Device Templates** section of the **Installed Templates** list. To create a new template library for function templates, select the **Function Templates** node in the **Installed Templates** list and click the **Create Library** button. **Result**: A new template library with a default name is added at the bottom of the **Function Templates** section of the **Installed Templates** list.
3	Rename the new template library as stated above and fill it with device or function templates by using for example the copy and paste operations described below.

Saving Template Libraries as File

The template libraries that contain device or function templates are EcoStruxure Machine Expert-specific XML files.

To provide them for use on other PCs, proceed as follows:

Step	Action
1	Select the template library you want to export in the **Installed Templates** list.
2	Click the **Save As File...** button.

Step	Action
3	In the **Save File** dialog box, navigate to the folder where you want to save the template library file.
4	Transfer the template library file to the other PC and install it by using the **Template Repository**.

Copy and Paste Operations for Template Libraries

The **Template Repository** dialog box also supports the copy and paste operation for template libraries.

To copy a template library with the device or function template it contains, select the respective item in the **Installed Templates** list and click the **Copy** button.

Now select the **Device Templates** or **Function Templates** node, and click the **Paste** button to insert a copy of this template library with a default name in the **Installed Templates** list.

Replace the default name by a name of your choice.

Copy and Paste Operations for Templates

The **Template Repository** dialog box supports the copy and paste operation for device or function templates.

To copy a device or function template, select the respective item from below a template library node in the **Installed Templates** list and click the **Copy** button.

You can now paste the template into a template library if the library is not write-protected.

A library can only be pasted into a library of the same kind.

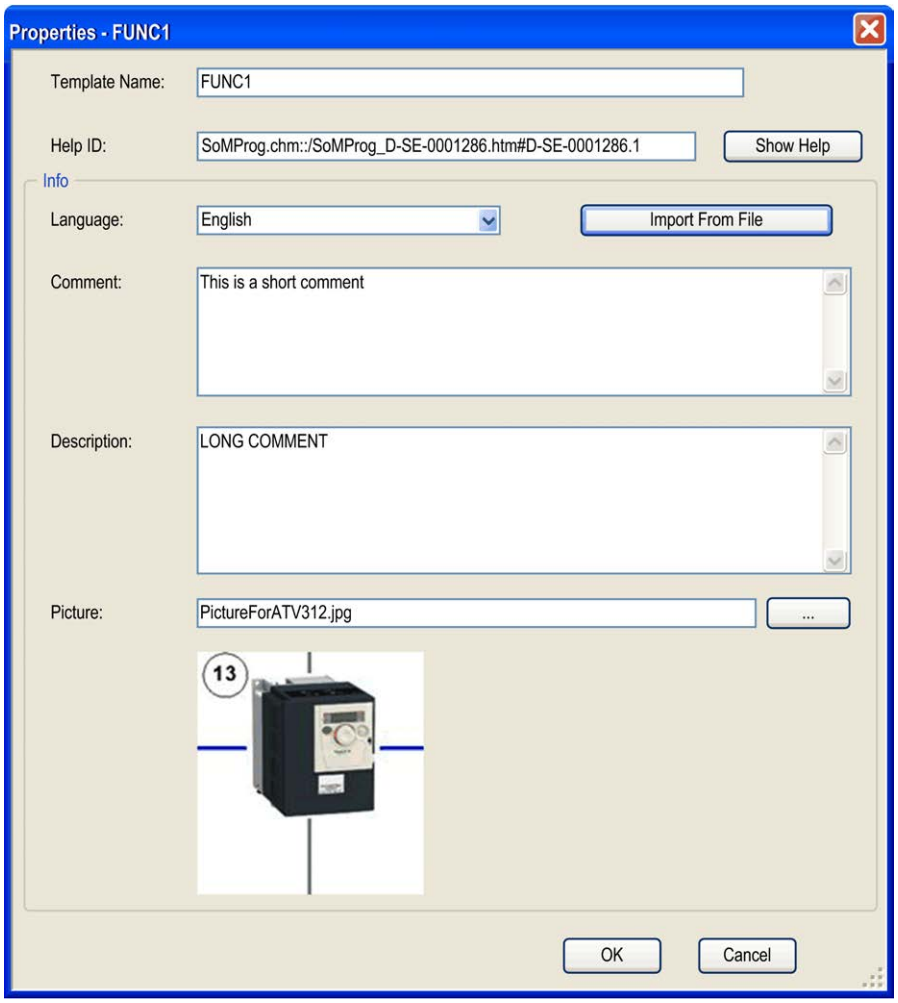
Replace the default name, if you wish, by a name of your choice.

Adding Further Information for Templates or Template Libraries

The **Template Repository** dialog box allows you to enter further information for templates or template libraries.

To add further information, select a library or template library in the **Installed Templates** list and click the **Properties...** button.

The **Properties** dialog box for the selected library or template library is displayed.



If the selected library or template library is not write-protected, the **Properties** dialog box contains the following parameters that you can edit, along with their corresponding buttons:

Element		**Description**
Template Name / **Library Name** box		Indicates the name of the library or template library these properties apply to. To change the name, click this box and adapt the name according to your requirements.
Help ID box		For Schneider Electric templates or template libraries, contains the reference to the respective description in the online help. If there is an online help document available for your templates, you can enter a full reference to its location in the online help or a keyword corresponding to an index in the online help.
Show Help button		Opens the online help document specified in the **Help ID** box or the index of the online help searching for the keyword specified in the **Help ID** box.
Info section		–
	Language list	Contains the languages that are available for the graphical user interface of EcoStruxure Machine Expert. If you select a language, the content of the language-dependent elements **Comment**, **Description**, and **Picture** is displayed in the selected language. If no language-specific content is available, the default language **English** is displayed.
	Import From File button	Displays a standard **Open** dialog box. It allows you to browse for an XML file that contains the localized content of the language-dependent elements **Comment**, **Description**, and **Picture**. The structure of this XML file must follow the structure indicated in the example, page 769.
	Comment box	Allows you to enter a short text (for example to provide an overview of the contents and purpose of the selected library or template library). This text is indicated as a tooltip when you select template libraries.

Element		Description
	Description box	Allows you to enter a long text (for example to provide a detailed description of the contents and purpose of the selected library or template library.
	Picture parameter **...** button	Allows you to enter a path to a language-specific picture. You can also click the **...** button to browse for the graphic file. Supported graphic formats: • Bitmap: **.bmp* • JPEG: **.jpg* • Graphics interchange format: **.gif* • Icon: **.ico* After the picture has been specified, it will be displayed in the **Properties** dialog box. If you click the **OK** button, the picture is embedded in the template.

The check box **Read-Only** is only available for template libraries to indicate whether the selected template library is in read-only status. It is not possible to change the status of the template library here.

Localization of Language-Dependent Elements

You can localize the content of the language-dependent elements **Comment**, **Description**, and **Picture** by importing an XML file with the following structure:

```
<?xml version="1.0" encoding="UTF-8"?>
<TemplateProperties>
<HelpId>SoMProg.chm::/SoMProg_D-SE-0001286.htm#D-SE-
0001286.1</HelpId>
<PropertySet languageId = "en">
<Comment>This is a short description</Comment>
<Description>This is a long description</Description>
<ImageFile>PictureEnglish.jpg</ImageFile>
</PropertySet>
<PropertySet languageId = "de">
<Comment>Kurze Beschreibung</Comment>
<Description>Lange Beschreibung</Description>
<ImageFile>PictureGerman.jpg</ImageFile>
</PropertySet>
</TemplateProperties>
```

Managing Device Templates

What's in This Chapter

Managing Device Templates..... 770

Managing Device Templates

Facts of Device Templates

General Information on the Usage of Terms

The following description applies to field devices as well as to I/O modules even though only the term field device is used to increase readability.

Content of Device Templates

Device templates are related to a specific field device or I/O module. They contain the following information:

- Fieldbus configuration
- Control logic (controller programming) (optional)
- Visualization elements (visualization programming) (optional)

Using Device Templates

Device templates are stored in template libraries. Each template library contains the definition for several device templates that have a common base (are related to motor control, for example).

You can select them and adapt them to the requirements of your individual projects in order to create new pre-configured and ready to use field devices.

Creating New Device Templates

To make your already configured field devices reusable for any project, save them as device templates. This also includes the controller programming and visualization linked to this field device.

Versions of Device Templates

During the creation of a device template, a verification is performed whether the device description for the device to be created actually exists. If it does not, the device is automatically updated to the latest version if a later version exists.

Adding Devices from Template

Overview

Device templates are related to a specific fieldbus device. They contain the following information:

- Fieldbus device configuration

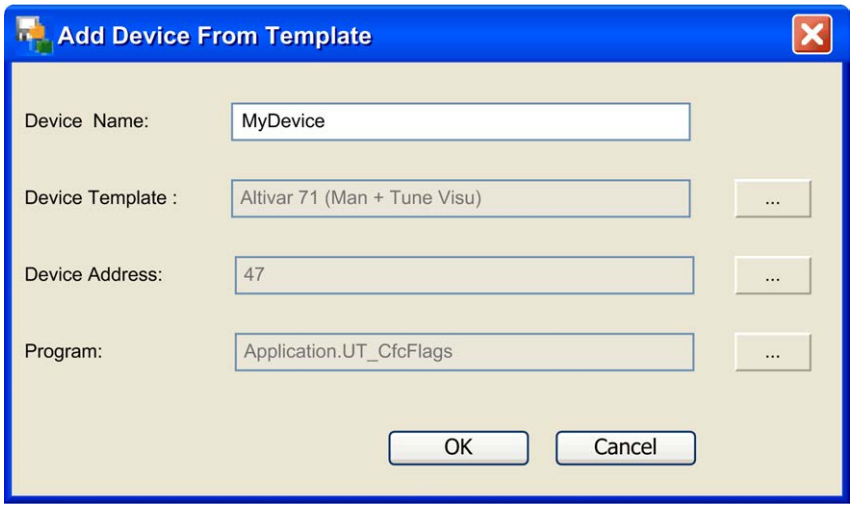
- Control logic (controller programming) (optional)
- Visualization elements (visualization programming) (optional)

You can create your own device templates from your project. For details, refer to the *Steps to Create a Device Template* chapter, page 776.

Add Device from Template

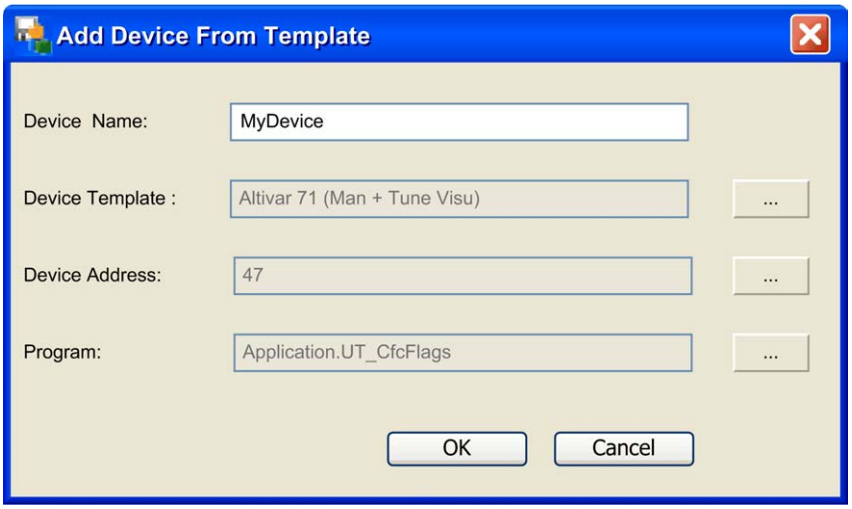
You can add a device from a device template in two different ways:

- Creating a device using a device template by drag-and-drop:

Step	Action
1	Open the **Devices & Modules** view of the hardware catalog.
2	At the bottom of the **Devices & Modules** view, activate the option **Device Template**. **Result**: The templates of field devices are displayed in the **Devices & Modules** view.
3	Select an entry in the **Devices & Modules** view, drag it to the **Devices tree**, and drop it at a suitable subnode of a controller. **Remark**: Suitable subnodes are highlighted. **Result**: The **Add Device From Template** dialog box displays. 
4	In the **Add Device From Template** dialog box, set the **Device Name** as well as the **Device Address** if the fieldbus requires numerical addresses. In case the device templates includes control logic, select the program (POU) in which the control logic is inserted.
5	Click the **OK** button. **Result**: The device is created and parameterized according to the selected device template including the optional visualization screens and control logic.

- Creating a device using a device template via contextual menu:

Step	Action
1	Open the **Devices tree**.
2	Right-click field device manager, and execute the command **Add Device From Template** from the contextual menu. **Result**: The **Add Device From Template** dialog box displays.

Step	Action
	
3	In the **Add Device From Template** dialog box, select the **Device Template** to be used, and set the **Device Name** as well as the **Device Address** if the fieldbus requires numerical addresses. In case the device templates includes control logic, select the program (POU) in which the control logic is inserted.
4	Click the **OK** button. **Result**: The device is created and parameterized according to the selected device template including the optional visualization screens and control logic.

NOTE: The undo / redo function is not available for the process of creating field devices.

Naming of Objects

In order to avoid naming conflicts if the same device template is used as a basis for creating different field devices, the following naming conventions are applied to the field devices and the associated objects (FB, visualization, and variables):

If the name of the original object...	**Then ...**
Case 1:	
contains the name of the original field device,	this part of the object is replaced by the name of the new field device that is created.
Example:	
The device template for the field device `ATV1` contains a variable `Var_ATV1_Input`.	For a new device `Axis1` being created with this device template, the new variable is correspondingly named `Var_Axis1_Input`.
Case 2:	
does not contain the name of the original device,	the name of the new device plus an underscore are inserted in the original name to form a unique new name.
Example:	
The device template for the field device `ATV1` contains a variable `Var_Input1`.	For a new device `Axis1` being created with this device template, the new variable is correspondingly named `Axis1_Var_Input1`.

Creating a Device Template on the Basis of Field Devices or I/O Modules

Overview

You can create device templates based on field devices or I/O modules. The following description applies to field devices as well as to I/O modules even though only the term field device is used to increase readability.

The following paragraphs list:

- The criteria that must be fulfilled in order to save a field device or I/O module, including logic and visualization, as device template;
- The information that is saved in the device template.

Prerequisites for Field Devices

The field devices must meet the following criteria in order to be saved as device templates:

- Field devices must be linked to the fieldbusses listed in the Supported Fieldbusses list, page 764;
- The device type must be installed in the **Device Repository**.

Prerequisites for I/O Modules

Only the supported I/O modules can be saved as device templates, page 764.

Prerequisites for the Application

You can only create templates from correct applications. Correct means that no errors are detected during the **Build** process.

Prerequisites for Including Control Logic into a Template

In order to include control logic into a template, the control logic must contain one or more code sections that exchange data with the field device. This control logic must be executed (added to a task or called by another program). Otherwise, it is not taken into account when executing the **Generate Code** command.

Device Information Saved in Device Templates

The following information of field devices is saved in device templates:

- Device configuration
- I/O mapping of the field device
- Visualizations that are suitable for the field device
- Control logic exchanging data with the field device

Visualizations Suitable for Creating Device Templates

Overview

Each device template can be associated with 1 or more Logic Builder visualizations. The supported types of visualizations are described as follows.

Supported Visualizations

Both types of visualizations are supported:

- Plain visualizations
- Modular visualizations using frames

Visualizations using frames have a better flexibility and modularity.

Plain Visualizations

Visualizations without frames are based on a single visualization object, created for the I/O device.

The data of the I/O device is referenced within the properties of the visual elements. When you create a new device based on this device template, the variables are directly replaced in the properties of the visual elements.

Visualizations Using Frames

A visualization using frames is built from a main screen that can be embedded with other visualizations, using a number of smaller visualizations to be combined like modules in predefined areas of the main screen (frames).

In the main screen, a frame-object is placed like a rectangular object as the container. You can assign another visualization to such a container.

The embedded visualization can then be used with an interface to access visual elements internally.

For more information, refer to the part **Software > Programming and Configuring > CODESYS Visualization** of the online help.

To use embedded visualizations for device templates, define an interface that includes definitions of all variables related to the connection to the I/O device or function block for each visualization module. When you create a new device based on this device template, all placeholders of the embedded visualizations are adapted according to the created I/O device name.

NOTE: All the visualizations using frames and the function blocks linked to the specific I/O device must be defined in a library so that they can be found.

Further Information on Integrating Control Logic into Device Templates

Overview

You can include control logic into a device template if the logic contains one or more code sections that exchange data with the field device in one of the following ways:

- A code section uses a new variable that is defined in the I/O mapping of the field device.
- A code section and the I/O mapping of the field device use a common variable that is defined in a GVL or a controller program contained by the application to which the code section belongs.

  NOTE: If you use structures or arrays, verify that they are only related to a single field device.
- A code section and the field device use a fix device-specific variable (for example the axis-ref variables used with the Altivar or Lexium drives).

Interconnected Calls of Code Sections

Code sections consist of a sequence of interconnected calls of function blocks, functions, and operators.

If one of the following relationships exists between the individual calls, they are considered as being connected:

- a graphical connection exists between the individual calls in CFC, FBD, and LD
- a variable is connected to the output of the one call and the input of the other call
- One call uses the parameter of the other call

Individually Selecting Function Blocks

You can individually select the function blocks that are included in those code sections that exchange data with the field device to be included in the device template. This allows you to create different device templates providing different functions for the same field device.

NOTE: The function block type must be defined in a library.

Including Expressions into Device Templates

The expressions, as well as the variables used in these expressions that are connected to the parameters of a function block, function or operator are automatically saved in the device template.

General Practices for the Creation of Control Logic

If you create a device template in one IEC language, but instantiate it within a different IEC language, the result may be a difference in execution. Therefore, it is best to keep such control logic as simple as possible.

By keeping the logic as simple as possible, the code sections will likely work identically even if they are created in different IEC languages.

NOTE: For complex control logic, you should rather create a function template.

Practices for the Creation of Control Logic in FBD / LD

Avoid edge detection elements because they do not exist in other IEC languages.

If possible, use `R_TRIG` or `F_TRIG` function blocks instead.

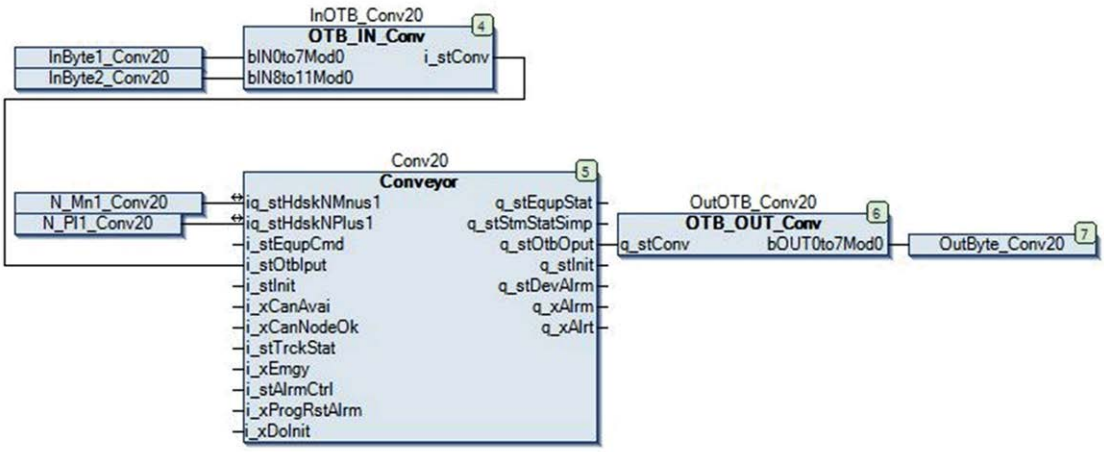
Practices for the Creation of Control Logic in CFC

Use the command **Execution Order > Order By Data Flow** to order the CFC elements belonging to the same code section according to their position in the data flow. This provides a better compatibility with other IEC languages.

Provide space (in horizontal direction) between the individual CFC elements because, due to renaming, the names of variables are extended when a new device is created from a template.

Control Logic Example

The following figure shows a typical example of a code section for an Advantys OTB distributed I/O device in a conveying application:



The code section consists of the following function blocks:

Name	Type	Function
`InOTB_Conv20`	Input block	Converting data coming from the OTB into the format required by the control block
`Conv20`	Control block	Processing data
`OutOTB_Conv20`	Output block	Converting data coming from the control block into the format required by the OTB

The variables `InByte1_Conv20`, `InByte2_Conv20` and `OutByte_Conv20` are defined in the I/O mapping of the OTB. Therefore, the code section exchanges data with the OTB device. It can thus become part of the device template.

Steps to Create a Device Template

Overview

The following paragraphs list the steps that have to be performed in order to save field devices meeting the criteria stated in *Creating a Device Template on the Basis of Field Devices*, page 773.

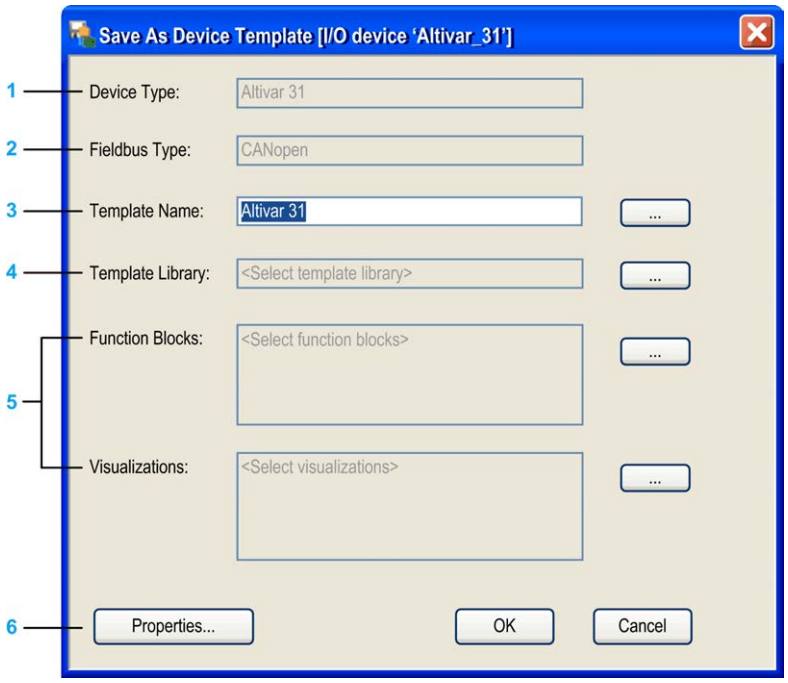
Steps for Saving a Field Device as Template

To save an already existing field device as device template, proceed as follows:

Step	Action
1	Right-click the field device you want to save as device template in the **Devices tree**.
2	Select the command **Save As Device Template** from the contextual menu. **Result**: The application is automatically built. After the built process has been successfully completed, the **Save as Device Template** dialog box will be displayed.
3	Define the new device template in the **Save as Device Template** dialog box as stated below.
4	Click **OK** to close the **Save as Device Template** dialog box and to create your new device template.

Save As Device Template Dialog Box

The **Save As Device Template** dialog box contains the following parameters:



1 indicates the type of the field device on which the device template is based

2 indicates the fieldbus type of the field device

3 the name of the device template that will be created (initially the name of the original field device)

4 select the template library the device template will be added to

5 select function blocks and visualizations that should be saved with the device template

6 Properties button to add further information to the device template

Defining a Name for the New Device Template

Use the text box **Template Name** to define a name for your device template.

By default, this text box includes the name of the selected field device.

You can either type the name of your choice directly into this text box, or you can click the **...** button to select an existing device template from the lists if you want to overwrite this device template.

Selecting the Template Library

To select one of the previously installed or created template libraries in which the device template should be stored, proceed as follows:

Step	Action
1	In the **Save as Device Template** dialog box, click the **...** button right to the **Template Library** text box. **Result**: The **Select Template Library** dialog box will be displayed.
2	The **Select Template Library** dialog box displays all template libraries that have been installed for the current project or have been created. Write-protected template libraries are not displayed. To add the new device template to 1 of these template libraries, select the suitable entry and click **OK**.

Selecting the Function Blocks

To select the function block instances to be included into the device template, proceed as follows:

Step	Action
1	In the **Save as Device Template** dialog box, click the **...** button to the right of the **Function Blocks** text box. **Result**: The **Select Function Block** dialog box will be displayed. The **Select Function Block** dialog box displays all function block instances contained by the control logic of the field device, page 774.
2	Select the check box of an individual function block to select it for the device template. Or select the check box of a root node to select all elements below this node.
3	Click the **OK** button.

Selecting the Visualizations

To select the visualizations to be included into the field device, proceed as follows:

Step	Action
1	In the **Save as Device Template** dialog box, click the **...** button to the right of the **Visualizations** text box. **Result**: The **Select Visualizations** dialog box will be displayed. The **Select Visualizations** dialog box displays those visualizations that are linked with the field device or with one of the selected function blocks.
2	Select the check box of an individual visualization to select it for the device template. Or select the check box of a root node to select all elements below this node.
3	Click the **OK** button.

Adding Further Information to the New Device Template

To add further information to the new device template, click the **Properties...** button. The **Properties** dialog box opens. It allows you to enter further information for the device template. Since the dialog box is identical for device templates and template libraries, see the description in the Adding Further Information for Templates or Template Libraries chapter, page 767.

Managing Function Templates

What's in This Chapter

Managing Function Templates 779

Managing Function Templates

Facts of Function Templates

Content of Function Templates

Function templates represent dedicated control and visualization functionality that are associated with an application function.

A function template may include the following elements:

- One or several IEC programs, functions, or function blocks
- One or several user-defined data types (DUT)
- One or several field devices or I/O modules that are being used by the application function
- One or several visualizations that are being used to visualize the application function
- One or several global variable lists
- One or several global variables that may be shared with other application functions
- One or several text lists and image pools
- One or several traces
- One or several CAM tables (for CANmotion)
- One or several I/O variables to be mapped on an I/O channel
- One or several template parameters

Using Function Templates

Already available function templates are saved in template libraries. Each template library contains the definition for several function templates that have a common base (for example, all are related to packaging applications).

You can easily select them and adapt them to the requirements of your individual projects in order to create new ready to use application functions.

Creating New Function Templates

To make your already created application function reusable for any project, you can save it as function template.

When you save the function template, decide in which template library it should be stored.

Versions of Function Templates

During the creation of a function template, a verification is performed whether the device description for the device to be created actually exists. If it does not, the device is automatically updated to the latest version if a later version exists.

Adding Functions from Template

Overview

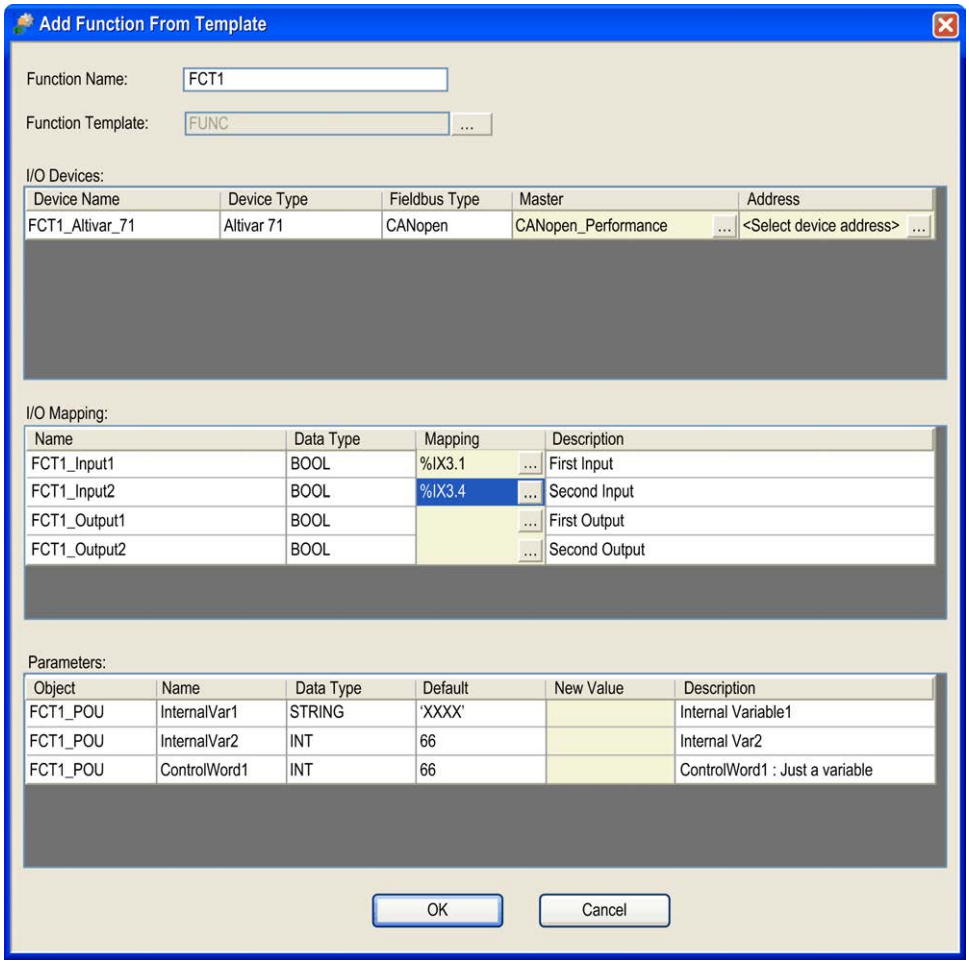
You can add an application function from a function template by right-clicking a suitable node in a navigator and executing the command **Add Function From Template** from the contextual menu.

Suitable nodes are:

- An application node in the **Applications tree**
- A folder node below an application node in the **Applications tree**
- A controller node in the **Functional tree**
- A functional module node in the **Functional tree**

As a result, the **Add Function From Template** dialog box opens.

Add Function From Template Dialog Box



The **Add Function From Template** dialog box provides the following elements to configure your function:

Element		**Description**
Function Name text box		Enter a name that is used for the new folder of this application and for the naming of the elements it contains.
Function Template		Click the **...** button and select a function template from the **Select Function Template** dialog box.
I/O Devices table		–
	Device Name	Contains the name of the future field device. You cannot change this name.

Element		Description
	Device Type	Indicates the type of the field device. You cannot edit this cell.
	Fieldbus Type	Indicates the fieldbus type of the field device. You cannot edit this cell.
	Master	Contains the fieldbus master to which the field device is connected. If there are several masters, you can select the master of your choice from the list.
	Address	Initially empty. For field devices on fieldbusses that require numerical addresses (Modbus serial line and CANopen), click the **...** button right to the field and assign the address of your choice.
I/O Mapping table		Lists the I/O variables that are part of the function template. It allows you to map them to the I/O channels of existing devices and modules.
	Name	Contains the name of the I/O variable that has to be mapped on an I/O channel.
	Data Type	Indicates the data type of the I/O channel to which the I/O variable was originally mapped.
	Mapping	Click the **...** button to open the **Select I/O Mapping** dialog box. It allows you to select an I/O channel on which you can map the selected variable. After the variable has been mapped to an I/O channel, this **Mapping** field contains the input or output address of the I/O channel on which the variable is mapped.
	Description	Contains a description of the I/O variable.
Parameters table		Lists the template parameters included in the function template.
	Object	Indicates the name of the GVL or program in which the variable is defined. You cannot edit this field.
	Name	Contains the name of the variable. You cannot edit this cell.
	Data Type	Indicates the data type of the variable. You cannot edit this cell.
	Default	Indicates the default value of the variable. This is the initial value of the variable when the template was created. You cannot edit this cell.
	New Value	Edit this cell if you want to assign a new value to the variable. If you leave this cell empty, the **Default** value is used for this variable. Enter a value that is valid for the given data type.
	Description	Contains a description of the variable.
OK button		Confirm your settings by clicking the **OK** button. **Result**: The settings are verified for correctness and the new application function is inserted as separate node below the **Application** node or an error detection message is displayed.

Select I/O Mapping Dialog Box

The **Select I/O Mapping** dialog box is used to map a variable selected in the **Add Function From Template** dialog box to an I/O channel.

It displays the available I/O channels in a tree structure, similar to the **Devices tree**. The root node is the controller. Only those I/O channels are displayed whose data type fits to the data type of the new variable.

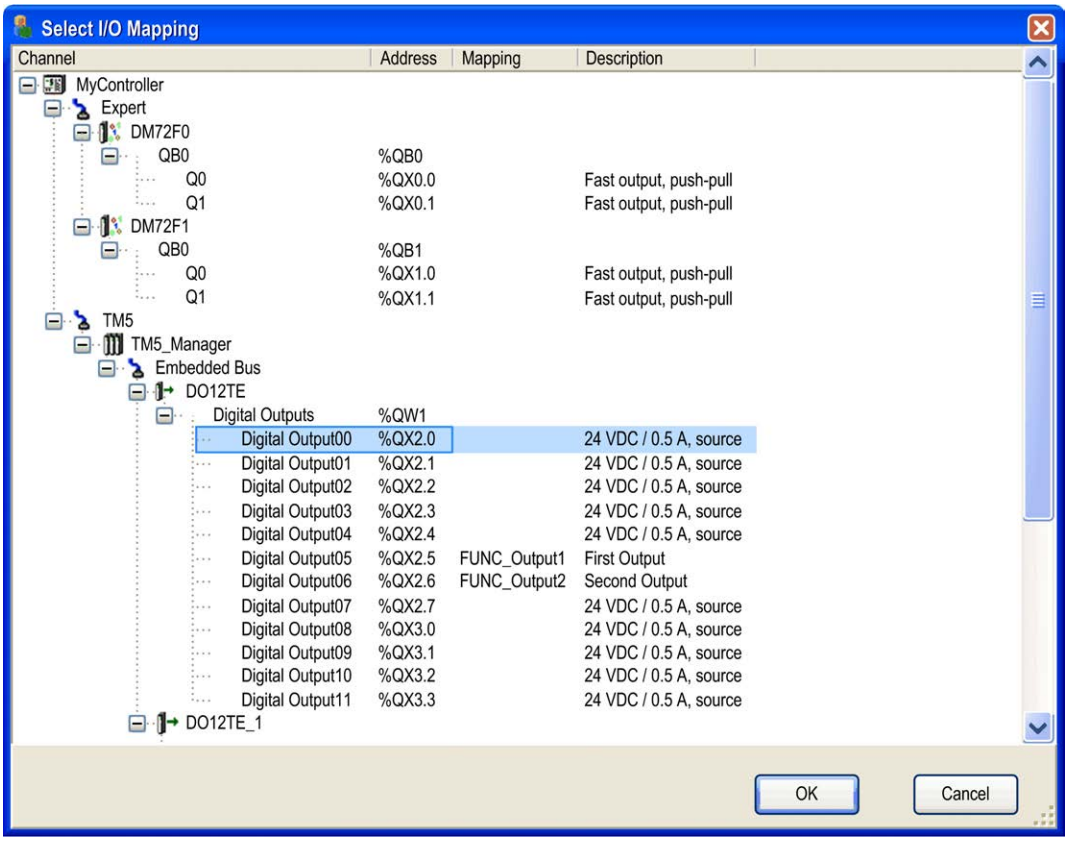
Two data types are compatible if they have identical type names or if they are elementary IEC data types of the same size.

Example:

UINT --> INT allowed

UDINT --> INT not allowed

Display the subnodes by clicking the plus signs.



The **Select I/O Mapping** dialog box contains the following columns:

Column	Description
Channel	Contains the tree structure. Each device is represented by the device name and the device icon. Each I/O channel is represented by the channel name.
Address	Contains the input / output address that corresponds to the I/O channel.
Mapping	Contains the I/O variable that is currently mapped on the I/O channel.
Description	Contains the description of the I/O channel.

Consider the following practices for mapping variables to I/O channels:

- Map all variables provided by the function template to I/O channels.
- You can map an I/O variable of a function template to an I/O channel that already has a mapping. The existing mapping is overwritten.
- Any mappings that lead to multiple assignments of variables on the same I/O channel are not allowed.

Objects Created

The function template creates the following objects in your project:

Object	Description
Root folder	A new folder is created under the **Application** node in the **Devices tree** that is named as defined in the **Function Name** text box.
Field devices	The field devices that are included in the function template are created using names that apply to the naming rules and are connected to the fieldbus master. The I/O mapping is automatically adjusted, if necessary.
Objects available as subnodes of the	The objects that are included in the function template are created below the root folder in the respective navigator (**Devices tree**, **Applications tree**, **Tools tree**) using names that apply to the naming rules. The properties of the objects are automatically adjusted.

Object	Description
root folder in the navigators	
Task configuration	The task configuration is adjusted as required by the function template.
Global variable lists	The global variable lists that are included in the function template are created below the root folder using names that apply to the naming rules.
External variables	Global variables whose global variable lists do not belong to the function template are restored in their original global variable list as follows: • If a global variable list with the original name does not already exist below the application, it is created automatically. • If a global variable with the original name does not already exist in this global variable list, it is created automatically. If the type of global variable is incorrect, an error detection message is displayed.
Persistent variables	Persistent variables are restored in the respective variable list of the application as follows: • If a persistent variable list does not already exist below the application, it is created automatically with its original name. • If a variable with the original name does not already exist in the persistent variable list, it is created automatically. If the type of persistent variable is incorrect, an error detection message is displayed.
External objects	Objects that are not included in the function template but are referenced by the function template (such as function blocks and DUTs) are handled as follows: • If the object does not exist, it is created • If the object already exists and has not been modified, it remains unchanged • If the object already exists and modifications are detected, an error is reported in the **Messages** view. To display further information on the detected modifications, click the entry in the **Messages** view.

Any objects that are created with the instantiation of the function template are listed in the **Messages** pane.

Naming of Objects

In order to avoid naming conflicts, if you instantiate the same function template several times on the same controller device, the following naming conventions are applied to the application functions and the associated objects:

If the name of the original object...	Then ...
Case 1:	
contains the name of the application function,	this part of the object is replaced by the name of the new application function that is created.
Example:	
The template original application function `Axis` contains a program `Axis_Init`.	For a new application function `Axis1` being created with this template, the new program is correspondingly named `Axis1_Init`.
Case 2:	
does not contain the name of the application function,	the name of the new application function plus an underscore are inserted in the original name to form a unique new name.
Example:	
The original application function `Axis` contains a program `InitProg`.	For a new application function `Axis1` being created with this function template, the new program is correspondingly named `Axis1_InitProg`.

NOTE: Use rather short names for your application functions so that they are completely displayed.

Application Functions as Basis for Function Templates

Overview

The following paragraphs list:

- The criteria that must be fulfilled in order to save an application function with its associated field devices, I/O modules, and visualizations as function template;
- The information that is saved in the function template.

Prerequisites for the Application

You can only create templates from correct applications. Correct means that no errors are detected during the build process.

Prerequisites for Saving an Application Function as Function Template

In order to save an application function as function template, it is a prerequisite that all programs of the application function are executed.

This means they must meet one of the following criteria:

- They must be added to a task.
- They must be called by another program.

Otherwise, they will not be considered when executing the **Generate Code** command.

I/O Variables in Function Templates

An I/O variable is a variable that is mapped on an I/O channel of a field device. It is saved in the function template if the following conditions apply:

- The I/O variable is used by any program or visualization that is included in the function template.
- The field device or I/O module to which the I/O variable is mapped cannot be included in the function template.

You can map an I/O variable that is saved in the function template on an existing I/O channel when an application function is created from the function template, page 780.

The I/O variable has a description that is displayed in the **Add Function From Template** dialog box.

This description is created as follows:

- If the I/O variable was newly created in the **I/O Mapping** tab of the device editor, page 148, the description is taken from the description of the I/O channel (this only applies if the original description has been changed).
- If the I/O variable is a reference to an existing variable, the description is taken from the comment of this variable.

Template Parameters

A template parameter is a variable with an adjustable initial value.

Example:When a device is used via a communication function block, then you have to assign the address of the device to this function block as an input parameter. To be able to set this address, connect a variable to the function block and define the variable as a template parameter.

A variable can become a template parameter if the following conditions apply:

- The variable is defined in a program or global variable list that is included in the function template.
- The variable has a simple data type (BOOL, any numeric data type, any STRING, alias types based on a simple data type).
- The initial value of the variable is explicitly defined as a literal value.

All variables that meet these conditions can be selected as template parameter when the function template is saved, page 788.

If a variable was selected as a template parameter, the initial value of this variable can be adjusted when a new application function is created from the function template, page 780.

Objects Saved in Function Templates

The following objects are saved in function templates:

- The objects that are available in the tree structure as subnodes below the **Application** node or below the **Functional Model** node.
- The field devices and I/O modules that are used.
- The function blocks, functions, or DUTs that are referenced by an object.

Steps to Create a Function Template

Overview

A function template can be created in two ways:

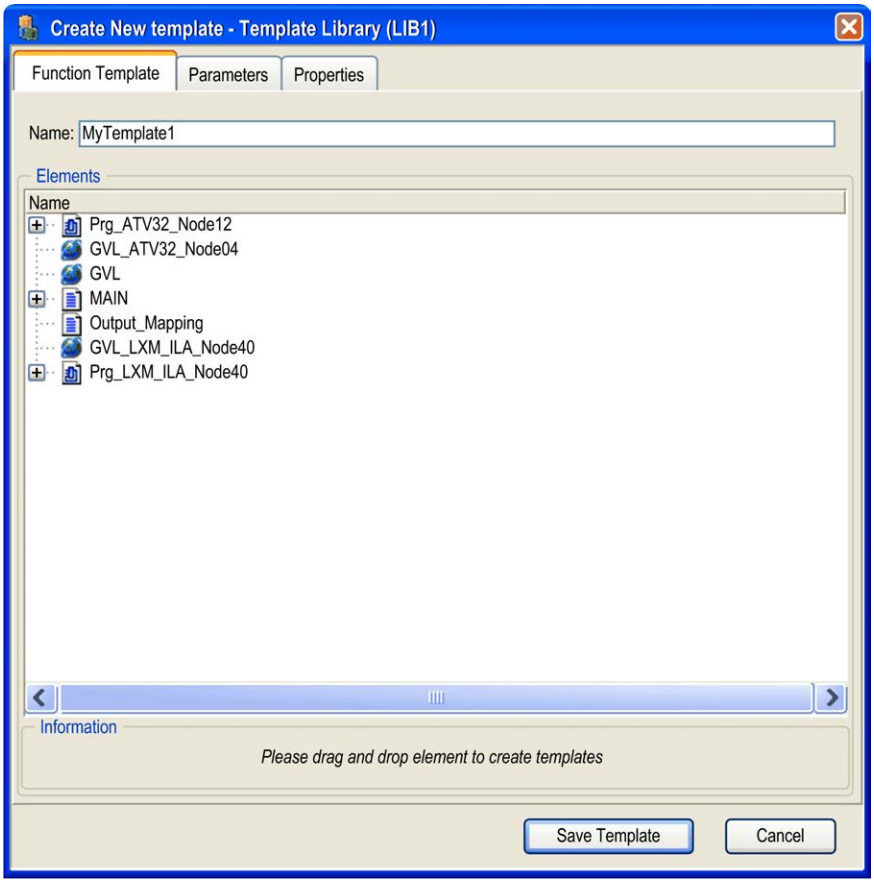
- From the **Macros** view using the **Create New Template** dialog box.
- From the **Applications tree** or **Functional tree** using the **Save as Function Template** dialog box.

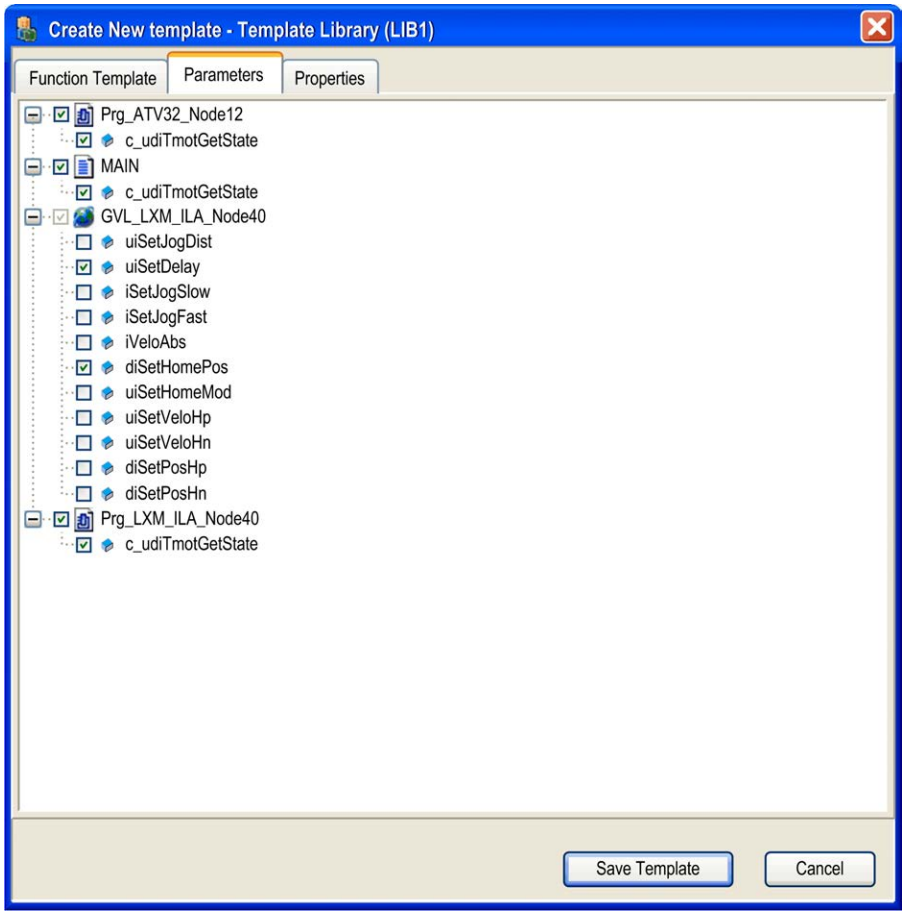
The following paragraphs list the steps that have to be performed in order to save already available application functions that meet the criteria stated in *Application Functions as Basis for Function Templates*, page 784 as function templates.

Procedure via Macros View

The procedure via **Macros** view allows you to create your own function template by dragging and dropping elements:

Step	Action
1	In the **Macros** view, expand the section **My Template**.
2	Select the **My Template** node, and click the green plus button. **Result:** A new node with the default name **LIB1** is inserted below the **MyTemplate** node.
3	Select the **LIB1** node and click the green plus button. **Result:** The **Create New Template** dialog box displays.

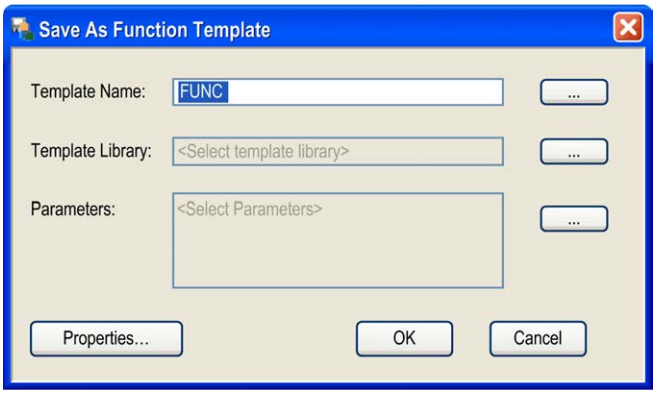
Step	Action
	
4	In the **Function Template** tab of the **Create New Template** dialog box, enter a **Name** for your function template. Drag the elements you want to include in the function template from the **Applications tree** to the **Elements** box of the **Function Template** tab. The elements listed in this box is inserted in your function template. **NOTE:** The elements must belong to the same application.
5	The **Parameters** tab of the **Create New Template** dialog box displays those variables that are included in the elements you selected in the **Function Template** tab.

Step	Action
	 From the list of variables, select those you want to declare as template parameters by selecting the check box of the variable or of a node.
6	The **Properties** tab of the **Create New Template** dialog box allows you to add further information to the function template. You can insert a link to the online help of this function template. The dialog box allows you to add further textual information that can be localized, and you can add a graphic illustrating this function template. For a description of these parameters, refer to the chapter *Adding Further Information for Templates or Template Libraries*, page 767.
7	Click the **Save Template** button.

Procedure via Applications tree or Functional tree

To save an already available application function as function template, proceed as follows:

Step	Action
1	Right-click a subfolder of an application node in the **Applications tree** or a functional model node in the **Functional tree**.
2	Execute the command **Save As Function Template** from the contextual menu.

Step	Action
	Result: The application is automatically built. After the built process has been successfully completed, the **Save As Function Template** dialog box is displayed. 
3	Define the new function template as stated below.
4	Click **OK** to close the **Save as Function Template** dialog box and to create your new function template. **Result**: It is verified that the function template can be created and a message is displayed that the function template has been created successfully or an error message is displayed.

Assigning a Template Name

In the **Template Name** text box of the **Save as Function Template** dialog box, define the name under which the function template is stored in the template library. By default, this text box contains the name of the folder that contains your application function in the **Applications tree** but you can adapt the name to your individual requirements.

Selecting the Template Library

To select one of the previously installed or created template libraries in which your new function template should be stored, proceed as follows:

Step	Action
1	In the **Save as Function Template** dialog box, click the **...** button next to the **Template Library** text box. **Result**: The **Select Template Library** dialog box is displayed.
2	The **Select Template Library** dialog box displays the template libraries that have been installed for the open project or have been created. Write-protected libraries are not displayed. To add your new function template to one of these template libraries, select the suitable entry and click **OK**.

Selecting Variables as Parameters

You can define variables of the function template as template parameters, page 785.

To define variables of the function template as template parameters, proceed as follows:

Step	Action
1	In the **Save as Function Template** dialog box, click the **...** button to the right of the **Parameters** text box. **Result**: The **Select Variables as Parameters** dialog box is displayed. It displays the variables that are defined in the selected application.
2	Select the check box of an individual variable to select it as template parameter for the function template. Or select the check box of a root node to select all elements below this node.
3	Click the **OK** button. **Result**: The selected variables are displayed in the **Parameters** text box of the **Save as Function Template** dialog box. They are displayed in the **Parameters** table of the **Add Function From Template** dialog box where you can assign **New Value**s for these parameters.

Overwriting an Existing Function Template

To overwrite an existing function template with the selected application function, proceed as follows:

Step	Action
1	In the **Save as Function Template** dialog box, click the **...** button right to the **Template Name** text box.
2	Browse to the already available function template you want to replace.
3	Select the function template you want to replace. **Result**: The name of this function template is inserted in the **Template Name** text box and the name of the template library where it is stored in is inserted in the **Template Library** text box.
4	Click **OK** to close the **Save as Function Template** dialog box and to replace the selected function template with the new application function.

Adding Further Information to the New Function Template

To add further information to the new function template, click the **Properties...** button. The **Properties** dialog box opens. It allows you to enter further information for the function template. Since the dialog box is identical for device templates and template libraries, see the description in the Adding Further Information for Templates or Template Libraries chapter, page 767.

Troubleshooting and FAQ

What’s in This Part

Generic - Troubleshooting and FAQ 791
Accessing Controllers - Troubleshooting and FAQ 796

Generic - Troubleshooting and FAQ

What's in This Chapter

Troubleshooting 791
Frequently Asked Questions 791

Troubleshooting

Log File

Overview

EcoStruxure Machine Expert automatically creates a log file. It provides information that is useful for debugging purposes. The size of the file is not restricted and there is no automatic removal or overwriting procedure.

The file is deleted only when EcoStruxure Machine Expert is uninstalled.

NOTE: Delete the file manually in suitable time intervals so that it does not use a large amount of disk space on your PC. The file is found in the temporary folder of Windows, defined by the variable *%TEMP%*.

Frequently Asked Questions

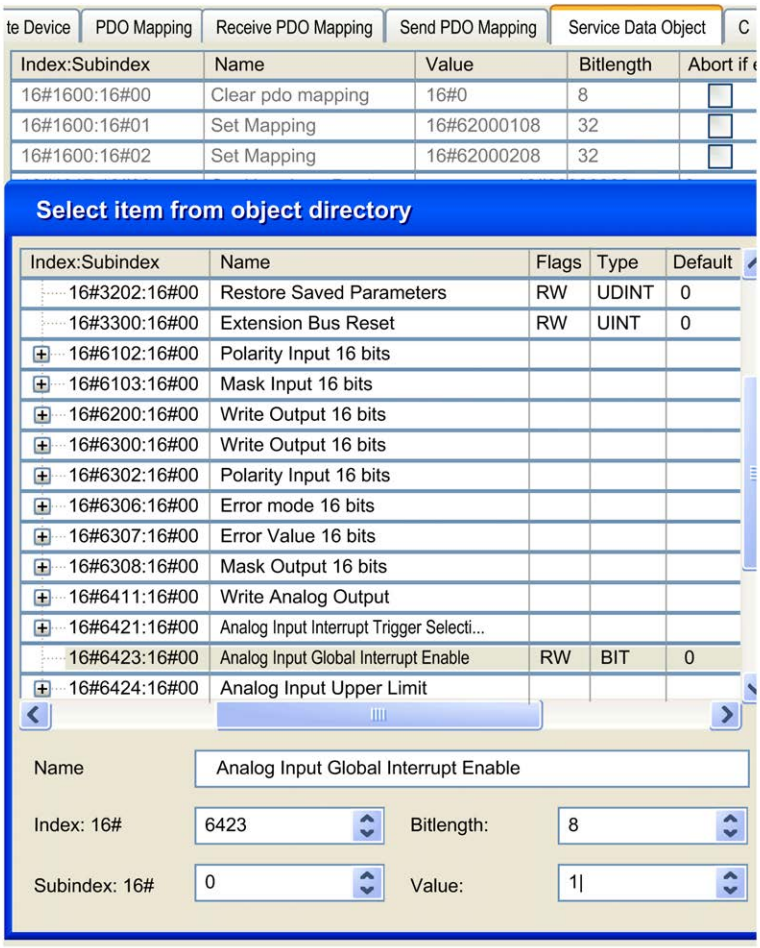
How Can I Enable and Configure Analog Inputs on CANopen?

Overview

This section provides instructions on enabling analog inputs according to the CANopen standard by setting the SDO (Service Data Object) 6423 to the value `1`.

Procedure

Step	Action
1	Double-click the node of your analog CANopen device in the **Devices tree**.
2	In the **CANopen Remote Device** tab of the editor, enable the option **Enable Expert Settings**. **Result**: Additional tabs are displayed and the **Service Data Object** tab is populated with information.

Step	Action
3	Open the **Service Data Object** tab and click the **New...** button. **Result**: The **Select item from object directory** dialog box is displayed. 
4	From the list of objects, select object **6423**, enter `1` as **Value**, and click **OK**. **Result**: Analog input transmission on the CANopen bus is activated. You can now configure parameters of the analog values as described in the hardware manual of your device.

Why is the Startup Performance Sometimes Slower?

Overview

Beside the PC configuration there are several other conditions which can increase the time EcoStruxure Machine Expert is consuming during startup:

Boot Phase	Startup Performance
first start after installation	On first start after installation, the software will generate its working environment on the PC. This is done only one time but has significant impact on the duration of the first startup.
first start after reboot	After rebooting the PC, the startup time can be longer than usual because Microsoft Windows consumes some time in the background to launch services that are needed to run EcoStruxure Machine Expert. This can have impact on the startup duration and cannot be avoided.
subsequent starts	Users experience better performance of the startup when the system has been started previously on the PC.

How Can I Manage Shortcuts and Menus?

Overview

The menus and shortcuts of the EcoStruxure Machine Expert software differ depending on the present state, that is, the window or editor that is open.

You can adapt the shortcuts and menus to your individual preferences or you can load the EcoStruxure Machine Expert or CODESYS standard shortcuts and menus as described in the following sections.

Customizing Shortcuts and Menus

You can adapt the shortcuts and menus to your individual preferences by using the **Tools > Customize** menu.

Restoring the EcoStruxure Machine Expert Standard Shortcuts and Menus

To restore the EcoStruxure Machine Expert standard shortcuts and menus (after you have customized them), proceed as follows:

Step	Action
1	Execute the **Customize** command from the **Tools** menu. **Result**: The **Customize** dialog box will be displayed.
2	In the **Customize** dialog box, click the **Load...** button. **Result**: The **Load Menu** dialog box will be displayed.
3	In the **Load Menu** dialog box, navigate to the folder *...\Program Files\Schneider Electric \EcoStruxure Machine Expert\Vx.x\LogicBuilder\Settings*, select the file *Standard.opt. menu*, and click **Open**. **Result**: The **Customize** dialog box now shows the standard EcoStruxure Machine Expert settings.
4	To load these standard settings to the graphical user interface, click **OK**.

Setting the Shortcuts and Menus to CODESYS Standard

To import the CODESYS shortcuts and menus to your graphical user interface, proceed as follows:

Step	Action
1	Execute the **Customize** command from the **Tools** menu. **Result**: The **Customize** dialog box will be displayed.
2	In the **Customize** dialog box, click the **Load** button. **Result**: The **Load Menu** dialog box will be displayed.
3	In the **Load Menu** dialog box, navigate to the folder *...\Program Files\Schneider Electric \EcoStruxure Machine Expert\Vx.x\LogicBuilder\Settings\OriginalCoDeSys*, select the file *Standard.opt.menu*, and click **Open**. **Result**: The **Customize** dialog box now shows the CODESYS settings.
4	To load these CODESYS settings to the graphical user interface, click **OK**.

NOTE: The menus and shortcuts of the EcoStruxure Machine Expert software differ, depending on the window or editor that is open.

Expanding Menus

EcoStruxure Machine Expert main menus and contextual menus can be displayed in a collapsed or full view. In the collapsed mode seldom used or disabled commands are hidden. After clicking the arrow menu item ▼ at the bottom of a menu, the corresponding menu expands, showing all its menu items.

For always showing the menus in the full viewing mode, activate the option **Always show full menus** in the **Tools > Options > Features** dialog box.

How Can I Increase the Build-Time Performance?

Good Practices

The list of good practices may help you to avoid a slow performance:

- Verify that the hardware of the PC meets the system requirements (see Introduction User Guide).
- Use a Solid State Drive (SSD) and verify that sufficient memory space is available. Contact your IT administration for further details.
- Consider uninstalling components that you do not need.
- If you are using Vijeo-Designer, consider to disable the automatic symbol export function. To achieve this, activate the option **Disable automatic symbol export** in the Logic Builder **Options > Vijeo-Designer** dialog box (see Menu Commands, Online Help).
- Consider activating the option **Disable undo after deleting a DTM (performance optimization)** in the Logic Builder **Options > FDT Options** dialog box (see Menu Commands, Online Help) if this function is not frequently used.
- Avoid large POUs. The size of POUs programmed in graphical languages like LD is not limited. Nevertheless, it is a good practice to call POUs in sequences instead of creating one large POU.

What Can I Do in Case of Issues with Modbus IOScanner on Serial Line?

Overview

This section provides instructions that may help you to solve issues that were detected when using Modbus IOScanner on a Serial Line.

Exception State of the Application

A Modbus IOScanner on a Serial Line is configured on your controller and one of the Modbus slave devices is disconnected.

If the application goes to exception state after the download or after a reset of the controller, proceed as follows:

Step	Action
1	Verify the integrity of your cable.
2	Verify that your cable is correctly connected between the controller and the Modbus Serial slave.
3	Reset your controller.

Error Detected While Using Modbus IOScanner on a Serial Line

If...	Then ...
an error is detected while using Modbus IOScanner on a Serial Line	the *xError* flag associated to the slave that has been detected as generating or causing the error is set to TRUE.
	the communication is NOT stopped (the controller still tries to connect the slave).
	the parameter *uiNumberOfCommunicatingSlaves* is decreased and *xAllSlavesOK* is set to FALSE.

After the communication to the slave has been re-established, a rising edge on the *xReset* entry of the slave is required:

- To reset *xError*.
- To update the values of *uiNumberOfCommunicatingSlaves*.
- To update the values of *xAllSlavesOK*.

What Can I Do If My Network Variables List (NVL) Communication Has Been Suspended?

Problem

The NVL communication has been suspended after an online change.

Solution

Restart the target controller.

Accessing Controllers - Troubleshooting and FAQ

What's in This Chapter

Troubleshooting: Accessing New Controllers 796
FAQ - What Can I Do in Case of Connection Problems With the Controller? 799

Troubleshooting: Accessing New Controllers

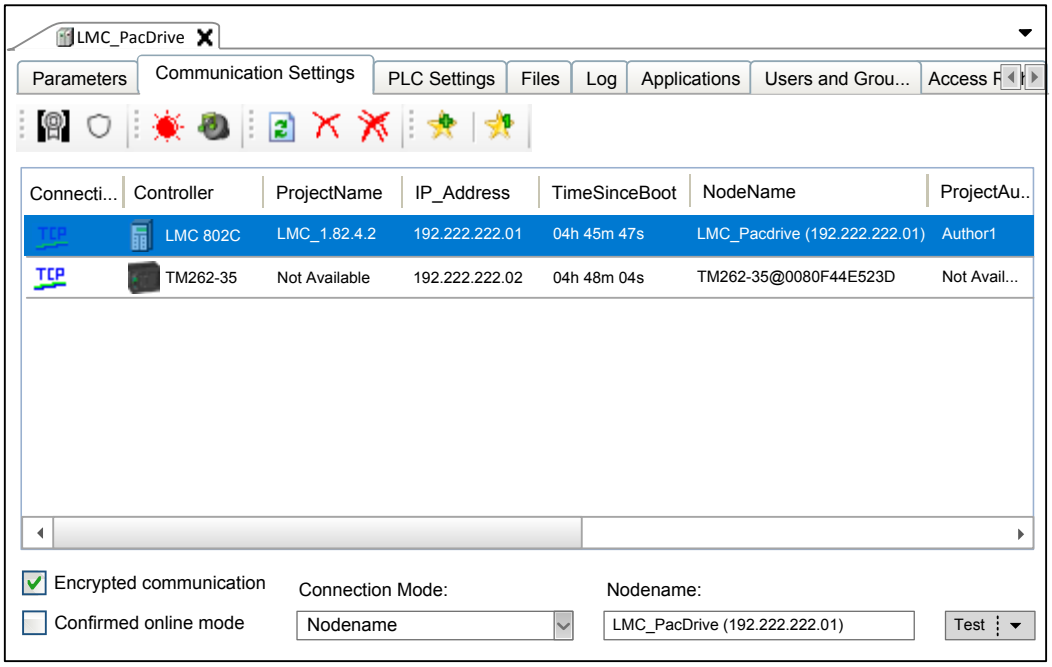
Accessing New Controllers

Overview

To access a new controller, adapt the network settings of the controller to the network of your EcoStruxure Machine Expert PC. This chapter provides a step-by-step example.

Example

This example shows the steps to access an M262 with IP address 192.168.1.33 from a PC with IP address 192.168.1.10 residing in subnet 255.255.255.0.

Step	Action
1	Connect the controller directly to the PC or to the network of the PC using an Ethernet cable.
2	In EcoStruxure Machine Expert, open the **Communication Settings** view of the device editor, page 87. **Result:** The M262 controller will be included in the list. 
3	To adapt the communication settings of the controller, right-click the controller in the **Communication Settings** list, and execute the command **Edit communication settings using** together with the suitable sub command from the contextual menu. **Result:** The **Edit communication settings** dialog box opens. For detailed information, refer to Configuring Communication Settings, page 95.
4	In the **Edit communication settings** dialog box, enter a free **IP address** available in your network. When configuring IP addresses, refer to the hazard message below.
5	Click **OK** to confirm the **Edit communication settings** dialog box.
6	In the **Communication Settings** view, connect to the controller.

Carefully manage the IP addresses because each device on the network requires a unique address. Having multiple devices with the same IP address can cause unintended operation of your network and associated equipment.

⚠WARNING

UNINTENDED EQUIPMENT OPERATION

- Verify that all devices have unique addresses.
- Obtain your IP address from your system administrator.
- Confirm that the device's IP address is unique before placing the system into service.
- Do not assign the same IP address to any other equipment on the network.
- Update the IP address after cloning any application that includes Ethernet communications to a unique address.

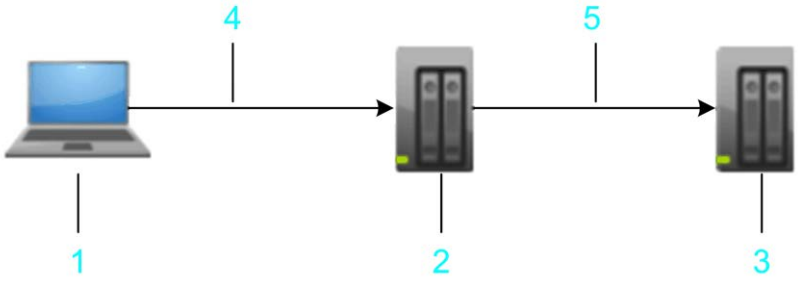
Failure to follow these instructions can result in death, serious injury, or equipment damage.

NOTE: Some controllers support a parameter that helps to prevent them from being remotely accessed (**RemoteCommunicationAccess** parameter of the LMC •0•C controllers.

Connecting via IP Address and Address Information

Overview

The used communication protocol offers a mechanism to connect to a controller independent of the type of connection. For example, this allows access to a target controller that is connected via Ethernet to another hop controller that is connected via USB to the PC itself.



1 PC

2 hop controller

3 target controller

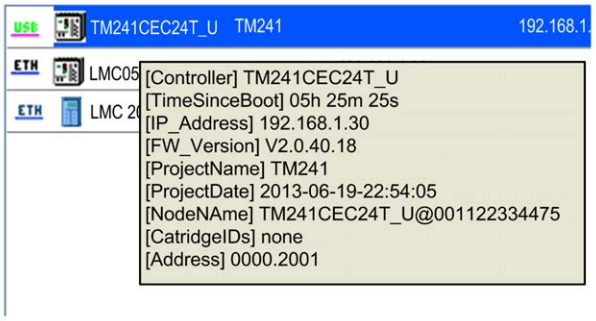
4 USB

5 Ethernet

Address Information

In the example, USB uses a different protocol. It is therefore normally not possible to use the IP address to address the target controller. Instead, the routing information is used that describes the way to connect to the target controller over 1 or more hops.

This routing information is displayed as a tooltip of an entry of the controller list (in the following example **[Address] 0000.2001**):



NOTE: Since this address only describes the way the controller is connected, it can change upon each modification of the local PCs or the network adapter settings of the hop controller. For example, upon activating or deactivating network adapters or upon starting/stopping services that use network adapters. The address to a specific target can differ from different sending PCs.

Nodename

Since the **Nodename** of the controller is a stable identifier in the system, it is used to identify the target.

If **IP Address** is selected as **Connection Mode**, it is tried to get the information from the **Nodename** itself. Some controllers (such as LMC •0•C) create the **Nodename** automatically including the IP address. You can also configure the **Nodename** by yourself (as described in the FAQ Why is the Controller not Listed in the Communication Settings View?, page 801) to enable the system to find a controller by its IP address. If the IP address is missing in the nodename, it is tried to get the IP address from a controller. But not all devices or their current firmware version support the service. In this case, use the **Connection Mode Nodename** to connect or set a device name that includes the IP address in brackets. For example **MyDevice (192.168.1.30)**.

FAQ - What Can I Do in Case of Connection Problems With the Controller?

FAQ - Why is a Connection to the Controller not Possible?

Why is a Connection to the Controller not Possible Even Though the IP Address Seems to Fit?

If you have set the IP address of the controller as described in the *Accessing New Controllers* chapter, page 796, and you still cannot connect to the controller, the reason can be the subnet mask. Since the communication protocol requires an identical subnet mask on both the sender and the receiver site, it may be possible that a ping to the controller is successful, but a connection cannot be established.

In order to solve this issue, proceed as follows:

Step	Action
1	Open the **Communication Settings** view of the device editor, page 87.
2	To adapt the communication settings of the controller, right-click the controller in the **Communication Settings** list, and execute the command **Edit communication settings using** together with the suitable sub command from the contextual menu. **Result:** The **Edit communication settings** dialog box opens.
3	Adapt the **Subnet mask** configured for the controller exactly to the subnet mask of your EcoStruxure Machine Expert PC. **Example:** Change **255.255.0.0** to **255.255.255.0**.

NOTE: After you have changed the **Connection Mode** in the **Communication Settings** dialog box, it may be required to perform the login procedure twice to gain access to the selected controller.

Why is a Login to a Controller not Possible?

For communications between an application (such as EcoStruxure Machine Expert, Controller Assistant) and a controller, a running EcoStruxure Machine Expert gateway is required. If you attempt to login to a controller, the application automatically starts the active EcoStruxure Machine Expert gateway. If EcoStruxure Machine Expert has not been started with (Windows) administrator rights, the start of the gateway cannot be executed.

In order to solve this issue, proceed as follows:

Step	Action
1	In the Windows notification area, verify whether the Gateway Management Console icon is displayed in red to indicate that the selected gateway is stopped: 
2	Right-click the Gateway Management Console icon, and execute the command **Start Gateway** from the contextual menu. **Result:** The selected gateway service is started.
3	In the Windows notification area, verify whether the Gateway Management Console icon is displayed in green to indicate that the selected gateway is running: 
4	Start another attempt to log in to the controller.

Why is the Controller not Listed in the Communication Settings Tab?

If you establish a connection between the controller and the EcoStruxure Machine Expert PC by using the classic mode, then the **Communication Settings** tab is displayed in classic mode, page 117.

If the controller of your choice is not displayed in the **Communication Settings** tab in classic mode, you can temporarily switch to the **Communication Settings** tab in controller selection mode, page 87 as follows:

Step	Action
1	Open the **Tools > Options > Device editor** dialog box (see Menu Commands, Online Help).
2	Select the **Communication page > Controller selection mode** setting and confirm by clicking **OK**.

Why is the Controller not Listed in the Communication Settings View?

If you do not find your controller in the list of the **Communication Settings** view in controller selection mode, page 87, the reason can be that the same **Nodename** is assigned to multiple devices. If the same **Nodename** is assigned to multiple devices, only one of these devices is listed in the list.

You must carefully manage the **Nodename** because each device on the network requires a unique **Nodename**. Having multiple devices with the same **Nodename** can cause unpredictable operation of your network and associated equipment.

⚠WARNING

UNINTENDED EQUIPMENT OPERATION

- Verify that all devices have unique Nodenames before placing the system into service.
- Update the Nodename after cloning any application that includes Ethernet communications to a unique Nodename.

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

To change the **Nodename** of a device, proceed as follows:

Step	Action
1	In EcoStruxure Machine Expert, open the **Communication Settings** in classic mode, page 117.
2	Right-click the device with a duplicate **Nodename** in the list and execute the commands **Set active path** and **Change device name** from the contextual menu. **Result:** The **Change device name** dialog box is displayed. 
3	In the **Change device name** dialog box, enter a unique **Nodename** in the **New** text box.
4	Click **OK** to confirm and to close the **Change device name** dialog box.
5	In the **Communication Settings** view, click the **Scan network** button to refresh the list of devices. **Result:** The second device with the same **Nodename** of the device you just have changed will now be displayed in the list.
6	Repeat steps 1...4 until you have eliminated duplicates.

NOTE: PacDrive LMC controllers automatically create a **Nodename** that is derived from the device name of the project and the IP address (for example, **MyLMC (192.168.1.30)**) after a project download. It is not possible to modify the **Nodename** directly, but by changing the IP address.

FAQ - Why has the Communication Between PC and Controller been Interrupted?

Why has the Communication Between PC and Controller been Interrupted?

It may be necessary to restart the gateway as follows:

Step	Action
1	Right-click the **Gateway Tray Application** icon in the Windows task bar  .
2	Execute the command **Restart Gateway** from the contextual menu.

Appendices

What’s in This Part

Network Communication 804
Python Script Language 809
Controller Feature Sets for Migration 876

Network Communication

What's in This Chapter

Network Topology 804
Addressing and Routing 804
Structure of Addresses 806

Network Topology

Overview

The control network is a system programmed to configure itself (address assignment) to support transparent communication media and to route packets between different networks. The routing mechanism lets any node in the network, that is, even nodes with low resources, reroute packets. So, large routing tables, complex calculations, or requests during runtime are avoided.

The control network is configured hierarchically, that is, each node has one parent node and an arbitrary number of children. A node without a parent is referred to as top-level node. Cycles are not permitted, that is, a control network has a tree structure.

Parent-child relationships arise from the specification of network segments. A network segment corresponds, for example, to a local Ethernet or a serial point-to-point connection. It distinguishes between the main network (mainnet) and the subnetworks (subnet). Each node has, at most, one main network, wherein it expects its parent. For each node, an arbitrary number of subnets can be configured. The node acts as parent for all of them.

If a network segment had been defined simultaneously as subnet of several nodes, the network would have several parents. However, the resulting configuration will be invalid, as each network segment is allowed to have one parent only.

Addressing and Routing

Overview

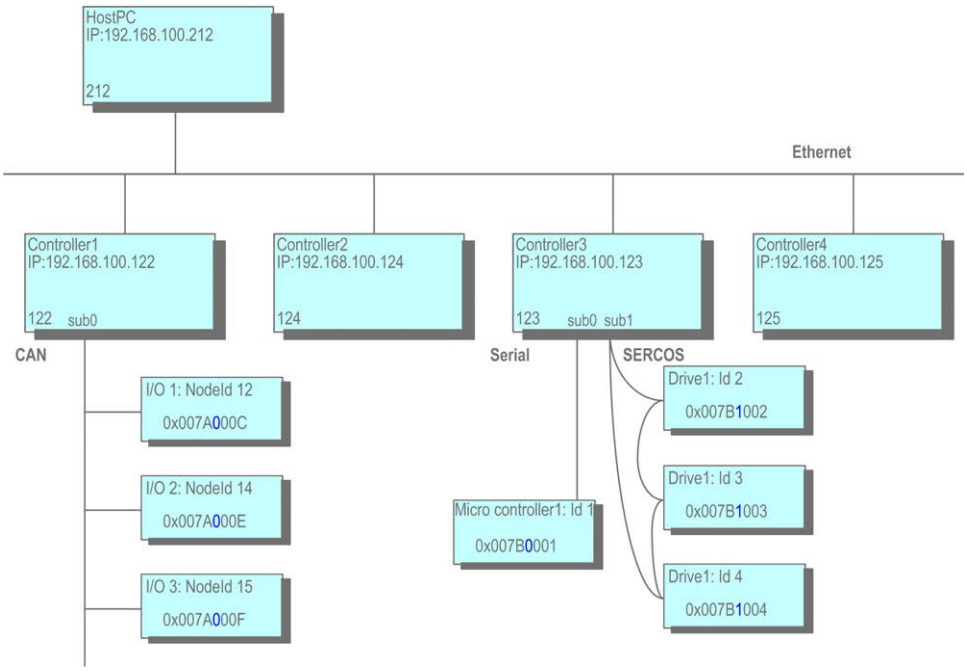
Addressing maps the topology of the control network to unique addresses. A node address, page 807 is built up hierarchically.

For each network connection, a local address identifying the node uniquely within its respective local network is allocated by the relevant block driver. For the entire node address, this local address is preceded by the subnet index the local network is assigned to by the parent. Furthermore, it must be preceded by the node address of its parent.

The length of the subnet index (in bit) is determined by the device, whereas the length of the local address is determined by the network type.

A node without a main network is a top-level node with address 0. A node with a main network that does not contain a parent is also a top-level node and will be assigned to its local address in the main network.

Example: Main net and sub nets



In the example, the addresses of the child nodes are given in hexadecimal representation. The first 4 digits represent the address of the particular parent within the main net. For example, 0x007A=122 for PLC1. The next byte (displayed in blue) is reserved for the subnet index and this is followed by the local address, for example, C=12 for node ID 12.

Due to the structuring of the address, the routing algorithm can be kept relatively lean. For example, no routing tables are necessary. Information is required locally: on the own address and on the address of the parent node.

Thereon, a node may properly handle data packets.

- If the target address equals the address of the current node, it is determined as receiver.
- If the target address starts with the address of the current node, the packet is intended for a child or descendant of the node and has to be forwarded.
- Else, the receiver is not a descendant of the current node. The packet has to be forwarded to the own parent.

Relative Addressing

Relative addressing is a special feature. Relative addresses, page 807 do not contain the node number of the receiver node, but directly describe the path from the sender to the receiver. The principle is similar to a relative path in the file system: The address consists of the number of steps the packet has to move up, that is, to the next respective parent, and the subsequent path down to the target node.

The advantage of relative addressing is that 2 nodes within the same subtree are able to continue the communication when the entire subtree is moved to another position within the overall control network. While the absolute node addresses will change due to such a relocation, the relative addresses are still valid.

Determination of Addresses

A node attempts to determine its own address as that coming from its parent or whether itself is a top-level node. For this purpose, a node will send an address determination via a broadcast message to its main network during boot-up. As long as this message is not responded to, the node considers itself to be a top-

level node, although it will continue to try to detect a parent node. A parent node will respond by an address notification. Thereon, the node will complete its own address and pass it to the subnets.

Address determination can be executed at bootup or on request of the programming PC.

Structure of Addresses

Overview

Below is a detailed description on the structure of the following address types:

- Network Addresses, page 806
- Node Addresses, page 807
- Absolute and Relative Addresses, page 807
- Broadcast Addresses, page 808

Network Addresses

Network addresses represent a mapping of addresses of a network type (for example, IP addresses) to logical addresses within a control network. This mapping is handled by the respective block driver. Within an Ethernet with class C IP addresses, the first 3 bytes of the IP address are the same for all network devices. Therefore, the last 8 bits of the IP address suffice as a network address since they allow unambiguous mapping between the 2 addresses at the block driver.

A node has separate network addresses for each network connection. Different network connections may have the same network address since this address has to be unique only locally for each network connection.

Terminology: In general, the network address of a node without a statement of the network connection refers to the network address in the main network.

The length of a network address is specified in bits and can be chosen by the block driver as required. Within a network segment, the same length must be used for all nodes.

A network address is represented as an array of bytes with the following coding:

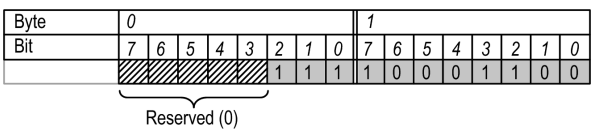
- Length of the network address: n bits
- Required bytes: b = (n + 7) DIV 8
- The (n MOD 8) lowest-order bits of the first byte and all remaining (n DIV 8) bytes are used for the network address.

Example - Network Address

Length: 11 bit

Address: 111 1000 1100

Example for network address coding



Node Addresses

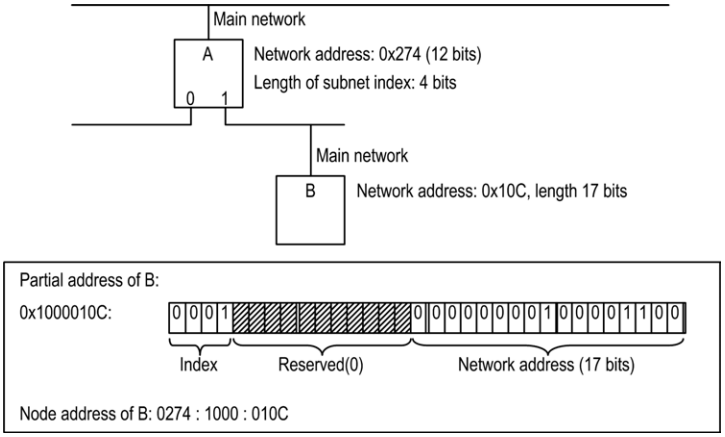
The node address indicates the absolute address of a node within a control network, and therefore, is unique within the whole tree. The address consists of up to 15 address components, each consisting of 2 bytes. The lower a node is located within the network hierarchy, the longer its address.

The node address consists of the partial addresses of all predecessors of the node and the node itself. Each partial address consists of one or several address components. The length is therefore always a multiple of 2. The partial address of a node is formed from the network address of the node in its main network and the subnet index of the main network in the parent node. The bits required for the subnet index are determined by the router of the parent node. Filler bits are inserted between the subnet index and the network address in order to ensure that the length of the partial address is a multiple of 2 bytes.

Special cases:

- Node has no main network: There is no subnet index nor a network address in the main network. In this case, the address is set to 0x0000.
- Node with main network but without parent: In this case, a subnet index with 0-bit length is assumed. The partial address corresponds to the network address, supplemented by filler bits if required.

Example - node address



The node address representation is always hexadecimal. The individual address components (2 bytes in each case) are separated by a colon (**:**). The bytes within a component display sequentially without a separator (see example above). Since this represents a byte array and not a 16-bit value, the components are not displayed in little-endian format. For manually entered addresses missing digits in an address component are filled with leading zeros from the left: 274 = 0274. To improve readability, the output should always include the leading zeros.

Absolute and Relative Addresses

Communication between 2 nodes can be based on relative or absolute addresses. Absolute addresses are identical to node addresses. Relative addresses specify a path from the sender to the receiver. They consist of an address offset and a descending path to the receiver.

The (negative) address offset describes the number of address components that a packet has to be handed upwards in the tree before it can be handed down again from a common parent. Since nodes can use partial addresses consisting of more than one address component, the number of parent nodes to be passed is always = the address offset. Therefore, the demarcation between parent nodes is no longer unambiguous. This is why the common initial part of the addresses of the communication partners is used as parent address. Each address component is counted as an upward step, irrespective of the actual parent nodes. Any error introduced by these assumptions can be detected by the respective parent node and must be handled correctly by the node.

On arrival at the common parent, the relative path (an array of address components) is then followed downwards in the normal way.

Formal: The node address of the receiver is formed by removing the last address offset components from the node address of the sender and appending the relative path to the remaining address.

Example

Within the example, a letter will represent an address component, whereas a point will separate the particular nodes. Since a node is allowed to have multiple address components, it is allowed to have multiple letters within the example.

Node A: a.bc.d.ef.g

Node B: a.bc.i.j.kl.m

- Address of the lowest common parent: a.bc
- Relative address from A to B: -4/i.j.kl.m (The number -4 results from the 4 components d, e, f, and g. Therefore the packet has to be raised).

The relative address has to be adjusted with each pass through an intermediate node. It is sufficient to adjust the address offset. This is always done by the parent node: If a node receives a packet from one of its subnets, the address offset is increased by the length of the address component of this subnet.

- If the new address offset is < 0, the packet must be forwarded to the parent node.
- If the address offset 1 ≥ 0, the packet must be forwarded to the child node of the local address of which is located at the position described by the address offset within the relative address. First, the address offset must be increased by the length of the local address of the child node to ensure that the node sees a correct address.

A special situation arises when the error described above occurs while determining the common parent. In this case, the address offset at the “real” common parent is negative, but the magnitude is greater than the length of the partial address of the subnet the packet originates from. The node must detect this case, calculate the local address of the next child node based on the address of the previous node and the length difference, and adapt the address offset such that the next node will see a correct relative address. Also, the address components themselves remain unchanged and only the address offset will be modified.

Broadcast Addresses

There are 2 types of broadcasts - global and local ones. A global broadcast is sent to all nodes within a control network. The empty node address (length 0) is reserved for this purpose.

Local broadcasts are sent to all devices of a network segment. For this purpose, all bits of the network address are set to 1. This is possible both in relative and in absolute addresses.

A block driver must be able to handle both broadcast addresses, that is, empty network addresses and network addresses with all bits set to 1, must be interpreted and sent as broadcast.

Python Script Language

What's in This Chapter

General Information 809
Schneider Electric Script Engine Examples 836
CODESYS Script Engine Examples 848
Managing Device User Rights Using the Scripting API 874

General Information

Introduction

Python Scripting

EcoStruxure Machine Expert includes a Python interpreter which can be used as a powerful tool to automate the development environment.

You can execute Python script files via the menu command **Tools > Scripting > Execute Script File...** in the main menu of the Logic Builder.

The documentation for Python scripting in EcoStruxure Machine Expert consists of several parts:

The chapter *Accessing the Python Interpreter* , page 811 describes how you can open the Python interpreter and how you can execute script files with it and how you can otherwise interact with it. The chapters *Using the Logic Builder Shell*, page 813, *Using the Logic Builder Scripting Immediate View*, page 819, and *Keyboard Commands in the Logic Builder Shell and the Scripting Immediate View*, page 821 provide more details on individual parts.

The sections *Schneider Electric Script Engine Examples*, page 836 and *CODESYS Script Engine Examples*, page 848 provide various examples for how to automate EcoStruxure Machine Expert with Python scripts. These examples are given to provide a common thread that shows the capabilities of the EcoStruxure Machine Expert-specific API (Application Programming Interface) available to Python scripts.

The chapters *Explore Python API (with dir() and inspectapi)*, page 822, *Best Practices*, page 827, and EcoStruxure Machine Expert *Scripting - Python API*, page 830 provide general hints on how to get started with Python scripting and how to explore the API provided.

The sections *Schneider Electric - Script Engine Class Library* and *Script Engine Plugin API Reference* provided in the *Programming* part of the online help provide the reference of the EcoStruxure Machine Expert-specific API available to Python scripts. You can use these members in the same manner as described in the aforementioned examples.

Compatibility Between Python V2.x and Python V3.x

The delivered Python interpreter is an implementation of IronPython based on Python version 2.7.

NOTE: There are some major changes in the Python language definition between version 2.x and version 3.x. While the Python interpreter delivered with EcoStruxure Machine Expert is based on Python V2.x, it is possible that you can write code compatible with the version 3.x implementation. This involves, for example, using the `from __future__ import print_function` statement.

For more information, refer to the websites

- http://wiki.python.org/
- http://docs.python.org/

Examples of new functions:

```
from __future__ import print_function
from __future__ import division
# New Python print syntax
print('Hello World!')
# Division
# Python 2 return an integer and rounds off
# Python 3 returns a float
print(17/3)
```

Coding Conventions

In order to harmonize and facilitate the work of different programmers on the same programming project, it makes sense to agree on a common programming style. EcoStruxure Machine Expert adheres to the *Style Guide for Python Code*. The delivered scripts and examples adhere to these conventions. For improved readability of code, it is a best practice that your own code adheres to the same conventions.

For further information, refer to the *Style Guide for Python Code* at http://www.python.org/dev/peps/pep-0008/.

Useful Links

For further information, refer to the following websites:

- Official Python webpage providing a tutorial and language references at http://docs.python.org/.
- Official Python blog at http://blog.python.org/.
- *Beginner's Guide to Python* at http://wiki.python.org/moin/BeginnersGuide.
- Official CODESYS forum providing examples and helpful information at http://forum.codesys.com/.
- IronPython interpreter at http://ironpython.codeplex.com/.
- Free Galileo Openbook (only available in German) at http://openbook.galileocomputing.de/python/.

Accessing the Python Interpreter

Overview

EcoStruxure Machine Expert can be automated with the Python script language.

The Python scripting capabilities are exposed in various ways. The table lists different entry points:

Where	Input type	Description / Use case	How to open
Logic Builder user interface	Interactive (REPL)	Interactive Python shell with a command-line interface, embedded in the user interface	Menu command **View > Scripting > Scripting Immediate** in the Logic Builder (also refer to the chapter *Using the* Logic Builder *Scripting Immediate View*, page 820).
	Python file (*.py)	Select a script file to be executed from within the user interface.	Menu command **Tools > Scripting > Execute Script File...** command in the Logic Builder (also refer to the chapter *Using the* Logic Builder *Scripting Immediate View*, page 819.
Stand-alone shell	REPL	Interactive Python shell with a command-line interface, running stand-alone (without the graphical user interface).	Open LogicBuilderShell.exe without command-line arguments (refer to the chapter *Using the Logic Builder Shell*, page 813).
	Python file (*.py)	Executes a Python script from the Windows command-line, from a batch file, or similar ways.	Open LogicBuilderShell.exe with a script file as a command-line argument (refer to the chapter *Using the Logic Builder Shell*, page 813).

The following sections provide an overview of how Python scripting integrates in EcoStruxure Machine Expert for different use cases.

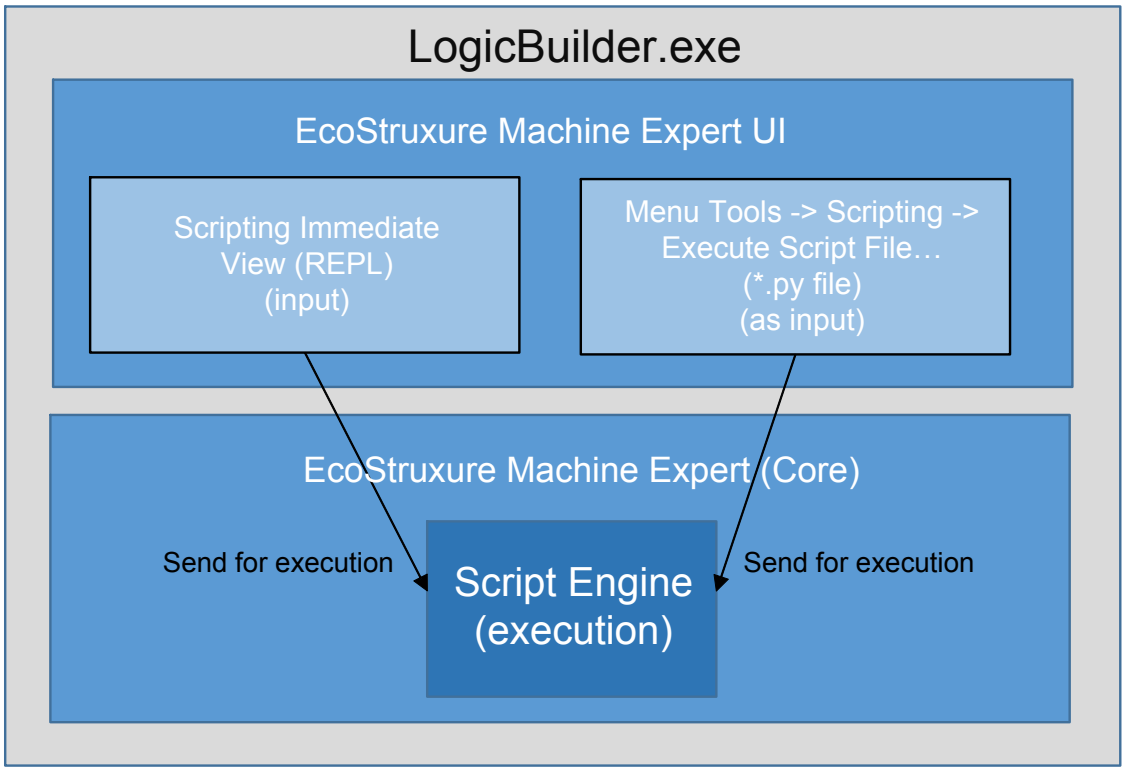
Logic Builder User Interface

The Logic Builder user interface allows you to execute script files via the **Tools > Scripting > Execute Script File...** command or executes scripting statements via the **Scripting Immediate** view

The **Scripting Immediate** view is a (Python) interpreter integrated in EcoStruxure Machine Expert allowing you, for example, to start functions.

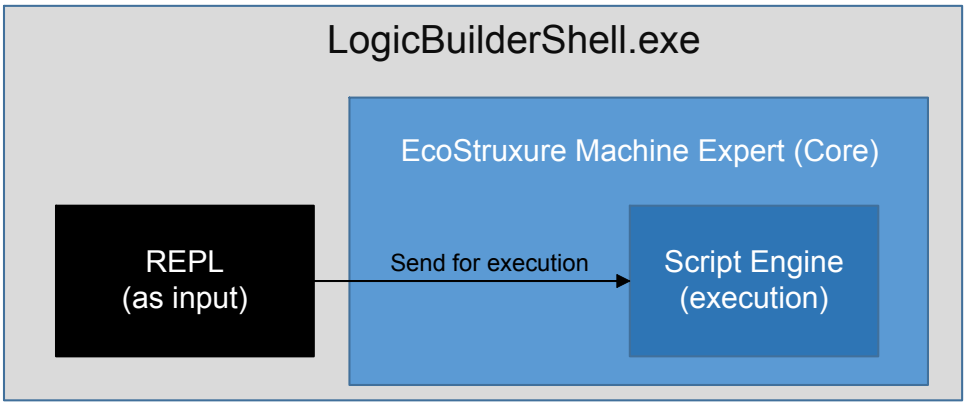
The **Scripting Immediate** view hosts an interpreter and is based on REPL principles.

The block diagram shows how the user interface is using the script engine to execute Python commands:



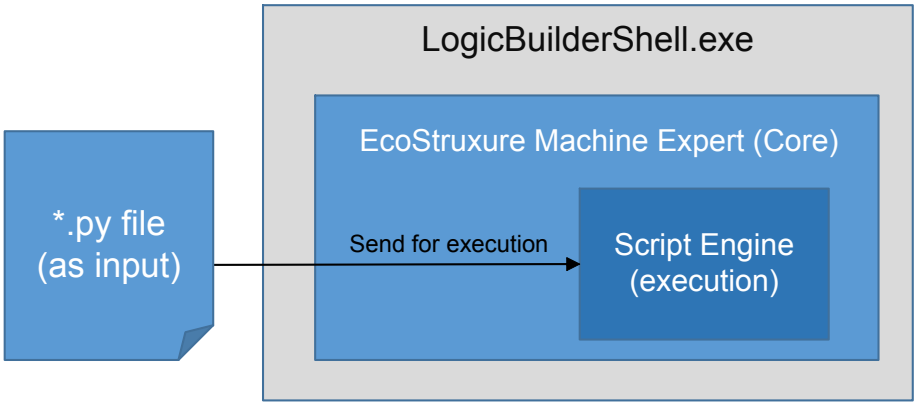
Execute Statements Via Interactive Shell

The block diagram shows how the interactive interpreter in a REPL-based shell is using the script engine to execute Python statements:



Script Execution Via Non-Interactive Shell

The block diagram shows how the (non-interactive) shell is using the script engine to execute scripts:



Using the Logic Builder Shell

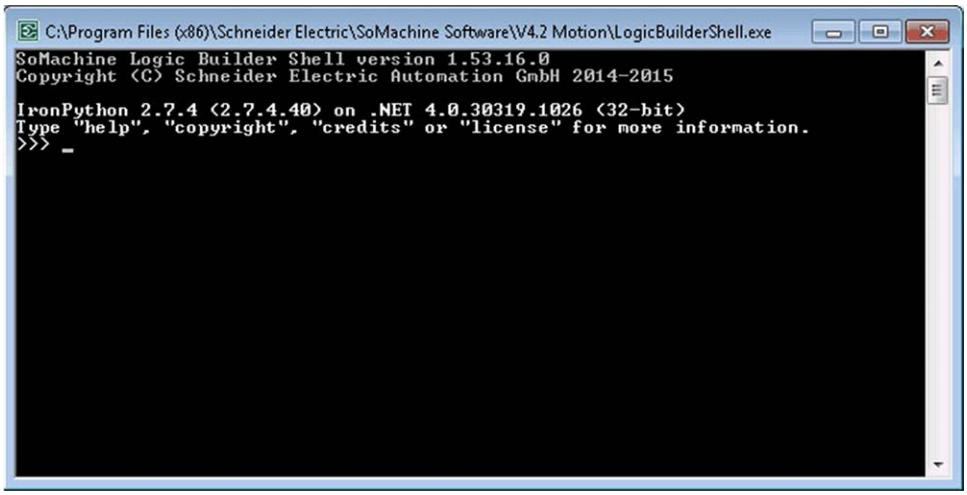
Starting the Logic Builder Shell

The LogicBuilderShell.exe is located in the installation directory of EcoStruxure Machine Expert.

The Logic Builder Shell provides a REPL that is based on the IronPython ipy.exe. This is the REPL of the standard IronPython package when installed on a PC. Hence, it provides the same functionalities, such as tab completion, Python script debugging support, and so on. Plus in addition, it allows access to the EcoStruxure Machine Expert Python API.

Double-click the LogicBuilderShell.exe to start the Logic Builder Shell (without arguments) in a console.

Result: The input prompt (>>>) is displayed:



Commands

At the input prompt (>>>), type your Python statements and press **Enter** to execute them.

The following sequence of commands provides an example on how to open a project, search a device node and print the name:

```
IronPython 2.7.4 (2.7.4.40) on .NET 4.0.30319.18444 (32-bit)
Type "help", "copyright", "credits" or "license" for more information.
>>> projectName = "C:\\temp\\MyProject.project"
>>> proj = projects.open(projectName)
>>> sercosNode = proj.find("SERCOSIII", True)[0]
>>> print(sercosNode.get_name(False))
SERCOSIII
>>>
```

Usage:
`LogicBuilderShell.exe [options] [file.py|- [arguments]]`

The `[options]` are defined in the list of command-line arguments below.

`[arguments]` is a placeholder for arguments that are passed to the script file.

Command-line arguments of the Logic Builder Shell:

Command-line argument	Description	Example
`-h / --help`	Prints the IronPython command-line help.	`LogicBuilderShell.exe --help` `LogicBuilderShell.exe -h`
`-m <module>`	Run library module as script.	`LogicBuilderShell.exe -m pdb MyScript.py`
`-i`	Inspect interactively after running script.	`LogicBuilderShell.exe -i MyScript.py`
`<Script File>`	Runs the specified Python script file.	`LogicBuilderShell.exe MyScript.py`
`--nologo`	Skips the display of the informational text of the Logic Builder during startup. Skips the display of the `...terminated` message during shutdown.	`LogicBuilderShell.exe --nologo`

You can combine different command-line arguments. For example, the `-i` argument can be combined with a specified Python script file and `-nologo`.

The list shows only a selection of IronPython command-line arguments. For a complete list, refer to the IronPython website and documentation.

Use Cases for the Logic Builder Shell

You can use the Logic Builder Shell in different scenarios. The following use cases show how it helps to improve the usability of working with EcoStruxure Machine Expert and the Python scripting language.

Use Case 1

When executing scripts in Continuous Integration (CI) systems - for example, to **Checkout** an EcoStruxure Machine Expert project from Subversion (SVN), compile the project and save as library - you need the script output printed to a console. With the LogicBuilderShell.exe as real console application, you can redirect the output or call it in a CI system (for example, Jenkins) where the output is fetched and added to a build log.

Example:
`LogicBuilderShell.exe MyScript.py > output.txt`

Use Case 2

LogicBuilderShell.exe allows you to use the built-in command-line debugging feature called `pdb` module.

Example:

```
LogicBuilderShell.exe -m pdb MyScript.py
```

Use Case 3

Test your Python scripts in no UI mode to verify that it is working. In console mode, you can enter input via the console. Keep in mind that reading input from the console is only possible in the LogicBuilderShell.exe. In the **Scripting Immediate** view, that is available in the user interface, no console input is possible. This has the effect that the statement `readline()` is ignored.

Example:

```
import sys
print("Please enter a text: ")
text = sys.stdin.readline()
print("Your entered text: " + text)
```

Use Case 4

Explore the Python and EcoStruxure Machine Expert API and documentation.

- Use the built-in `help()` feature to access the Python help via the IronPython shell. To achieve this, make sure that an Internet connection is available.

Example:

```
help()      # start the built-in help module
topics      # list all available topics
##### here you'll get the list of available topics #####
LIST        # dumps the help to work with lists.
##### here you'll get the help working with lists #####
quit        # leaf build-in help feature
```

- Use the built-in `dir()` feature to print available api functions.

Example:

```
dir()       # prints the list of defined variables and functions in current script
scope
##### here you'll get the list of defined variables or functions in current script
scope #####
import sys
dir(sys)    # prints the available functions defined in sys module
##### here you'll get the list of defined variables or functions in sys module
#####
```

- Use the `inspectapi` module to explore the EcoStruxure Machine Expert Python API. Due do technical restrictions, the built-in `dir()` feature of Python cannot print the API help for the EcoStruxure Machine Expert Python API. To achieve this, use the `inspectapi`.

Example:

```
inspectapi.dir(projects)  # prints the available API functions of "projects" which
provides access e.g. to open a project
##### here you'll get the Machine Expert API provided via "projects" variable
#####
```

Use Case 5

Use `-i` switch as argument passed to the LogicBuilderShell.exe in combination with a specified script to get the shell prompt at the end of the script run. This can help to inspect interactively after running a script, for example, to see the content of variables.

Benefits of the Logic Builder Shell

It is a best practice to use the LogicBuilderShell.exe to execute scripts instead of using LogicBuilder.exe to run scripts in no UI mode. Both ways are still possible but the command line of the LogicBuilderShell.exe provides a better usability and the Logic Builder Shell prints the output to the console.

Example of using the LogicBuilder.exe to run scripts in no UI mode:
```
LogicBuilder.exe –noui --runscript="<script file>"
```

Example of using the LogicBuilderShell.exe:
```
LogicBuilderShell.exe "<script file>"
```

Debugging Python Scripts with Command-Line Debugger Module `pdb`

The LogicBuilderShell.exe provides the built-in command line debugging feature `pdb` module (for more documentation, refer to https://docs.python.org/2/library/pdb.html). This is a feature for experts and used to debug Python scripts or to find an issue. IronPython itself provides a module that can be loaded into a shell. When you start your own script via the `pdb` function `run(…)`, you can debug the Python code in the console. The console application only provides text-based UI possibilities.

The module `pdb` defines an interactive source code debugger for Python programs. It provides the following features:

- Setting (conditional) breakpoints and single stepping at the source line level
- Inspection of stack frames
- Source code listing
- Evaluation of arbitrary Python code in the context of any stack frame

To start a Python script directly in `pdb`, use the following command-line syntax:
```
LogicBuilderShell.exe –m pdb MyScript.py
```

Selection of `pdb` debugger commands (refer to the Python help on the Internet for more details):

Debugger command	Command description
`h(elp) [command]`	Without argument, print the list of available commands. With a command as argument, print help about that command. `help pdb` displays the documentation file.
`r(un)`	Restart the debugged Python program.
`b(reak)`	With a line number argument, set a break at this position in the file. With a function argument, set a break at the first executable statement within that function. The line number may be prefixed with a file name and a colon to specify a breakpoint in another file (probably one that has not been loaded yet). The file is searched on `sys.path`. Each breakpoint is assigned a number to which the other breakpoint commands refer. If a second argument is present, it is an expression which must evaluate to TRUE before the breakpoint is honored. Without argument, list all breaks. For each breakpoint listed, the number of times that the breakpoint has been hit, the ignore count, and the associated condition, if any, are included.
`l(ist) [first[, last]]`	List source code for the file. Without arguments, list 11 lines around the current line or continue the previous listing. With one argument, list 11 lines around at that line. With two arguments, list the given range; if the second argument is less than the first, it is interpreted as a count.
`c(ontinue)`	Continue execution, only stop when a breakpoint is encountered.
`n(ext)`	Continue execution until the next line in the current function is reached or it returns. (The difference between `next` and `step` is that `step` stops inside a called function while `next` executes called functions at (nearly) full speed, only stopping at the next line in the current function.)
`s(tep)`	Execute the current line, stop at the first possible occasion (either in a function that is called or on the next line in the current function).
`u(ntil)`	Continue execution until the line with the line number greater than the current one is reached or when returning from the current frame.
`r(eturn)`	Continue execution until the current function returns.
`w(here)`	Print a stack trace, with the most recent frame at the bottom. An arrow indicates the current frame, which determines the context of most commands.
`q(uit)`	Quit from the debugger. The program being executed is aborted.

Debugging Example with `pdb`

Example Python script to be debugged:

```
print("Demonstration how to use Python pdb module to debug Python scripts")
# close open project
if projects.primary:
    print("Close project")
    projects.primary.close()
print("Open project")
projects.open("c:\\Temp\\MyProject.project")
sercosDevice = projects.primary.find("SERCOSIII", True)[0]
sercosDeviceName = sercosDevice.get_name(False)
print("Sercos device name: " + sercosDeviceName)
print("Close project")
projects.primary.close()
```

In each line in the following listing starting with `(pdb)`, a debugger command is executed, such as `list (l)` or `help` or `next (n)` or `quit (q)`. In between you can see the statement or its output which was executed.

```
LogicBuilderShell.exe –m pdb PythonDemonstration.py
Logic Builder Shell version 1.53.16.0
Copyright (C) Schneider Electric Automation GmbH 2014-2015
Demonstration how to use Python pdb module to debug Python scripts
> C:\temp\Pythondemonstration.py(5)<module>()
-> if projects.primary:
(Pdb) help
Documented commands (type help <topic>):
========================================
EOF    bt         cont      enable  jump  pp       run      unt
a      c          continue  exit    l     q        s        until
alias  cl         d         h       list  quit     step     up
args   clear      debug     help    n     r        tbreak   w
b      commands   disable   ignore  next  restart  u        whatis
break  condition  down      j       p     return   unalias  where
Miscellaneous help topics:
==========================
exec  pdb
Undocumented commands:
======================
retval  rv
(Pdb) l
  1
  2     print("Demonstration how to use Python pdb module to debug Python
scripts")
  3
  4     # close open project
  5  -> if projects.primary:
  6         print("Close project")
  7         projects.primary.close()
  8
  9     print("Open project")
 10     projects.open("c:\\Temp\\MyProject.project")
 11
(Pdb) n
> C:\temp\Pythondemonstration.py(9)<module>()
-> print("Open project")
(Pdb) n
Open project
> C:\temp\Pythondemonstration.py(10)<module>()
-> projects.open("c:\\Temp\\MyProject.project")
(Pdb) l
  5     if projects.primary:
  6         print("Close project")
  7         projects.primary.close()
  8
  9     print("Open project")
 10  -> projects.open("c:\\Temp\\MyProject.project")
 11
 12     sercosDevice = projects.primary.find("SERCOSIII", True)[0]
 13
 14     sercosDeviceName = sercosDevice.get_name(False)
 15
(Pdb) n
IOError: IOError(...oject'.")
> C:\temp\Pythondemonstration.py(10)<module>()
-> projects.open("c:\\Temp\\MyProject.project")
(Pdb) q
Logic Builder Shell terminated.
```

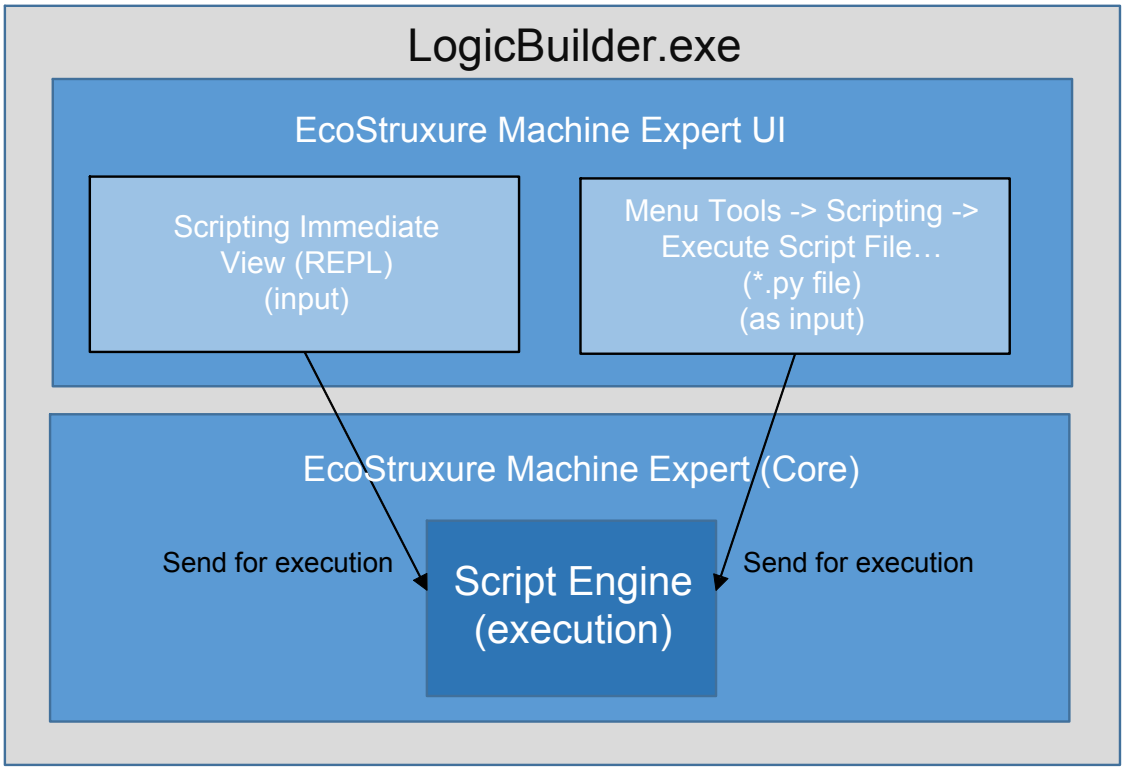
Using the Logic Builder Scripting Immediate View

Overview

When you start the Logic Builder with user interface via LogicBuilder.exe, then the following two commands are available to open two views dedicated to scripting purposes:

- **View > Scripting > Scripting Immediate**
- **View > Scripting > Scripting Watch**

Script execution in the EcoStruxure Machine Expert user interface via the **Execute Script File...** command or via the **Scripting Immediate** view:

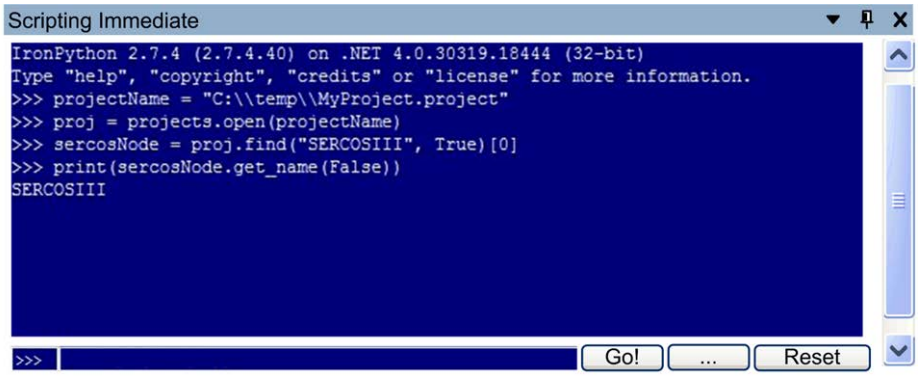


Scripting Immediate View

The **Scripting Immediate** view is a host for the same REPL as you can use in the Logic Builder Shell. It allows you to interact with EcoStruxure Machine Expert in a textual based interface in parallel to the user interface.

Execute the command **View > Scripting > Scripting Immediate** to open the **Scripting Immediate** view.

The figure provides an example of how to open a project, find a device object and print its name in the **Scripting Immediate** view.



The **Scripting Immediate** view allows you to enter Python statements and to execute them (using the REPL).

To run a Python script file, click the **...** button or enter the script name into the prompt textbox and click the **Go!** button.

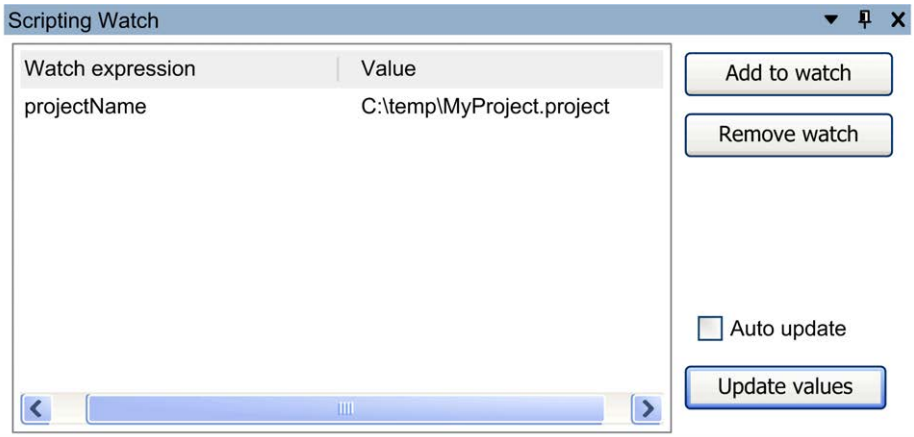


Click the **Reset** button to reset the **Scripting Immediate** view and to clear the current scope.

Scripting Watch View

The **Scripting Watch** view shows the content of the Python variables you have defined.

Execute the command **View > Scripting > Scripting Watch** to open the **Scripting Watch** view.



Elements of the **Scripting Watch** view:

Element	Description
Add to watch button	To add a watch expression, type the expression into the **Scripting Immediate** view prompt, and click the **Add to watch** button.
Remove watch button	To remove a watch expression, select an entry in the **Scripting Watch** view, and click the **Remove watch** button.
Auto update option	Select the **Auto update** option to enable the auto update function for expressions in the **Scripting Watch** view. **NOTE:** The process of refreshing values can lead to additional PC CPU load.
Update values button	The **Scripting Watch** view list is updated automatically each time a statement is executed in the **Scripting Immediate** view. To update the list manually, click the **Update values** button.

Example

Enter and execute the following statement in the **Scripting Immediate** view.
`projectName = "C:\\temp\\MyProject.project"`

Proceed as follows:

Step	Action
1	Enter `projectName` in the **Scripting Immediate** view.
2	Click the **Add to watch** button in the **Scripting Watch** view. **Result**: The variable `projectName` is displayed in the **Scripting Watch** list.

When you assign a different value to the variable `projectName`, the changed content is displayed.

Keyboard Commands in the Logic Builder Shell and the Scripting Immediate View

Overview

This chapter provides information on using the Logic Builder Shell started via the LogicBuilderShell.exe or the Logic Builder **Scripting Immediate** view.

Tab Completion / Intellisense

When you type Python statements, you can use the **tab** key on your keyboard to complete the entered statement or to iterate through the options.

Example:

Type the text `ins` and press the **tab** key. **Result**: The expression is competed to `inspectapi`.

Type the text `inspectapi.d` and press the **tab** key multiple times. The different functions of `inspectapi` starting with `d` are displayed.

Clear Current Entered Prompt Text

Type a Python statement and press the **ESC** key on your keyboard. The line is cleared.

Python Statement History

Press the **up** or **down** key on your keyboard to use the history of entered Python statements.

List Variables of Current Scope

Use the `dir()` function to display variables of the current scope.

`exit()` / `exit(…)` Function

Use the `exit()` / `exit(…)` function to leave the script and (if specified) to return the exit code for later usage.

Example:

Python script (`exit_code_test.py`)

```
# …
# Python statements
exit(5)
```

Running the script in the Windows command prompt (or batch file) and using the `exit` code:

```
LogicBuilderShell.exe exit_code_test.py
echo %ERRORLEVEL%
```

Result:

`5` is printed to the command prompt. You can use this `%ERRORLEVEL%` to control your batch flow.

Explore Python API (with `dir()` and `inspectapi`)

Overview

EcoStruxure Machine Expert provides a Python API that you can use for EcoStruxure Machine Expert scripting. When you write Python scripts, most times you only need names of functions or the list of required parameters and its names. You often also need the available functions of an object returned by an API function you have called. The following functions allow you to explore the EcoStruxure Machine Expert Python API to support you in developing Python scripts.

Built-in `dir(…)` function of Python

Without arguments, this function returns the list of names in the current local scope. With an argument, this function attempts to return a list of valid attributes for that object.

The default `dir()` mechanism of Python behaves differently with different types of objects, as it attempts to produce the most relevant, rather than complete information:

- If the object is a module object, the list contains the names of the attributes of the module.
- If the object is a type or class object, the list contains the names of its attributes, and recursively of the attributes of its bases.
- Otherwise, the list contains the names of the attributes of the object, the names of the attributes of its class, and recursively of the attributes of the base classes.

Example (in LogicBuilderShell.exe):

```
>>> import struct
>>> dir()   # show the names in the module namespace
['__builtins__', '__doc__', '__name__', 'struct']
>>> dir(struct)   # show the names in the struct module
['Struct', '__builtins__', '__doc__', '__file__', '__name__',
 '__package__', '_clearcache', 'calcsize', 'error', 'pack', 'pack_into',
 'unpack', 'unpack_from']
>>> class Shape(object):
        def __dir__(self):
            return ['area', 'perimeter', 'location']
>>> s = Shape()
>>> dir(s)
['area', 'perimeter', 'location']
```

`inspectapi` Functions

Since the `dir()` function does not list all EcoStruxure Machine Expert Python API elements, additional functions are provided to display them for Python script developers.

`inspectapi` is a Python module loaded automatically while starting the LogicBuilderShell.exe or the **Scripting Immediate** view. As a result, a variable/module `inspectapi` is available in the current scope. This variable is also listed by executing the `dir()` function.

The `inspectapi` variable/module provides the following functions:

Function name	Parameters	Description
`inspectapi.dir`	`obj`: Specify an object (for example, a variable) for which you get more information. (optional) `verbose`: The default value is FALSE. Set the parameter to TRUE to get verbose output.	This function provides a list of available EcoStruxure Machine Expert Python API members you can call. The fields, properties, events, and methods are listed.
`inspectapi.dir_events`		This function provides a list of available EcoStruxure Machine Expert Python API events you can call.
`inspectapi.dir_fields`		
`inspectapi.dir_methods`		
`inspectapi.dir_properties`		
`inspectapi.help`	`()`	This function opens the CODESYS *Scripting Engine* in the **Software > Programming and Configuring > CODESYS Scripting API** part of the online help. For further information, also refer to the Schneider Electric Script Engine User Guide Online Help.
	`obj`: Specify an object (for example, a variable) for which the online help will be displayed.	This function opens the online help of the object. • If there is no documentation for the object, the online help for the scripting engine opens. • If there are multiple help topics for the object (because there are extensions for the object or the object has multiple documented interfaces), a dialog box opens listing the relevant help topics. Double-click an entry to open the corresponding help topic. • If there is only one help topic for the object, the help topic opens directly.

Examples:

```
# To see the API of the inspect api itself
inspectapi.dir(inspectapi)
```

Output:

```
Members of type 'InspectAPI'
Methods:
  dir (obj, verbose = False)
  dir_events (obj, verbose = False)
  dir_fields (obj, verbose = False)
  dir_methods (obj, verbose = False)
  dir_properties (obj, verbose = False)
  help (obj = null)
# To see the API of projects variable
inspectapi.dir(projects)
```

Output:

```
Members of type 'ScriptProjects'
Properties:
  [get] all
  [get] primary
Methods:
  convert (stPath, stOutputPath, converter, bPrimary)
```

```
  convert (stPath, stOutputPath, converterGuid, bPrimary)
  create (stPath, bPrimary)
  get_all ()
  get_by_path (stPath)
  get_primary ()
  open (stPath, stPassword, bPrimary)
  open (stPath, encryption_password, session_user, session_password, bPrimary)
  open_archive (stArchiveFile, stProjectPath, bOverwrite, stPassword)
  open_archive (stArchiveFile, stProjectPath, bOverwrite, encryption_password,
session_user, session_password)
# open a project
proj = projects.open("c:\\temp\\MyScript.py ”)
# To see API of primary project
inspectapi.dir(projects.primary)
```

Output (truncated):

```
Members of type 'ScriptProject'
Properties:
  [set] active_application
  [get] dirty
  [get] handle
  [get] has_library_manager
  [get] has_project_info
  [get] is_root
  [get] library
  [get] path
  [get] primary
  [get] project
  [get] svn
  [get] user_management
Methods:
  add (stName, id, stModuleId)
  add (stName, iType, stId, stVersion, stModuleId)
  check_all_pool_objects ()
  clean_all ()
  close ()
  compare_to (projectFile)
  compare_to (projectFile, ignoreWhiteSpace, ignoreComments, ignoreProperties)
  ...
  logout ()
  save ()
  save_archive (stArchiveFile)
  save_archive (stPath, additional_categories)
  save_archive (stPath, comment, additional_categories)
  save_archive (stPath, additional_files, additional_categories)
  ....
  save_as (stPath, stPassword)
  save_as_compiled_library (destination_name)
  set_active_application (value)
  update ()
# search objects by name (e.g. a device with name SERCOSIII)
objs = projects.primary.find("SERCOSIII", True)
# extract object at index 0 from returned list (if found).
theObj = obj[0]
# see the available API of the found object SERCOSIII
inspectapi.dir(theObj)
```

Output (truncated):

```
Members of type 'ScriptObject'
Properties:
  [get] connectors
  [get] device_parameters
  [get] embedded_object_types
  [get] guid
  [get] handle
  [get] has_textual_declaration
  [get] has_textual_implementation
  [get] index
  [get] is_application
```

```
  [get] is_device
  [get] is_explicit_connector
  [get] is_folder
  [get] is_libman
  [get] is_project_info
  [get] is_root
  [get] is_task
  [get] parent
  [get] project
  [get] svn
  [get] type
Methods:
  add (stName, id, stModuleId)
  add (stName, iType, stId, stVersion, stModuleId)
  allowed_interfaces_at (index)
  can_convert (targetDeviceId, targetModuleId = null)
  can_convert (targetType, targetId, targetVersion, targetModuleId = null)
  convert (targetDeviceId, targetModuleId = null)
  convert (targetType, targetId, targetVersion, targetModuleId = null)
  create_folder (stFolderName)
  disable ()
  enable ()
  export_native (destination, includeChildren, profileName, reporter)
  export_xml (reporter, stPath, bRecursive)
  export_xml (reporter, stPath, bRecursive, bExportFolderStructure)
  export_xml (reporter, stPath, bRecursive, bExportFolderStructure, bPlainText)
  export_xml (stPath, bRecursive, bExportFolderStructure, bPlainText)
  find (namePath)
  find (stName, recursive)
  get_address ()
  get_all_parameters ()
  ...
  move (newParent, nNewIndex)
  plug (stName, id, stModuleId)
  plug (stName, iType, stId, stVersion, stModuleId)
  reboot_plc ()
  remove ()
  rename (stNewName)
  reset_diagnosis_messages ()
  set_communication_address (address)
  set_gateway_and_address (stGateway, stAddress)
  set_gateway_and_address (gateway, stAddress)
  set_parameter (parameter, value)
  set_parameter_iec_address (identifier, connectorId, iecAddress)
  set_parameter_io_variable_mapping (identifier, connectorId, variable,
createVariable = False)
  set_parameter_io_variable_mapping (identifier, connectorId, subElementIndex,
variable, createVariable = False)
  set_simulation_mode (bSimulation)
  unplug ()
  update (id, stModuleId)
  update (iType, stId, stVersion, stModuleId)
```

Executing Scripts

Overview

You can execute script files (*filename.py*), containing a sequence of script commands, from the user interface.

For further information on running scripts form the user interface, refer to the chapter *Script-Related Commands* (see Menu Commands, Online Help).

Batch Files

Frequently used commands

Command	Description
– `REM` or `::`	The line is a comment and will be ignored.
`cd`	Changes to another directory.
`echo off`	The commands will not be displayed. In order to prevent single commands from being displayed, insert an `@` character in front of the command.
`echo`	Displays a string or a variable on the programming console.
`set`	Declares a variable and assigns a value to this variable.
`>`	Writes the output to a file. If the file already exists, it will be overwritten.
`>>`	Appends the output to a file. If the file does not already exist, it will be created.

Application example:

```
@echo off
REM Go to the directory where EcoStruxure Machine Expert is
installed
cd "<Replace this with the path to the LogicBuilderShell.
exe, for example, C:\Program Files\Schneider Electric
\EcoStruxure Machine Expert\V2.0>"
REM Run LogicBuilderShell.exe with no graphical user
interface and the full path to the script
LogicBuilderShell.exe "<Replace this with the full file path
where the script is stored, for example, D:\MyScripts
\TestScript.py>"
pause
```

Best Practices

Indentation (Tabs and Spaces)

Blocks in Python are created by indentation of codes of block (unlike, for example, `END_IF` statements in structured text, or curly brackets `{" "}` in the C programming language).

Examples:

C Programming language	Python programming language

`int factorial(int x)`
`  {`
`  if (x > 1)`
`  return x* factorial(x – 1);`
`  else`
`  return 1;`
`  }`
	`def factorial(x):`
	`  if x > 1:`
	`  return x * factorial(x – 1)`
	`  else:`
	`  return 1`

Note that when mixing tabs and spaces for indentation: In Python, tab characters are internally replaced by 8 spaces. Since numerous editors by default use 4 spaces, this can lead to code errors that are difficult to find. Source code blocks appear to have the same size of the indent but actually the indentation is different.

Therefore, care should be taken when using copy and paste commands.

To help to avoid this, configure your editor in such a way that it automatically replaces tab characters by spaces.

Case Sensitivity

Python is a case-sensitive language (in contrast to, for example, structured text). Therefore, `MyVar` and `MYVAR` refer to two different variables.

Encoding Path Names

Python uses the backslash `\` character to encode special characters, for example, the sequence `\n` encodes the newline character.

Backslash characters are also commonly used to encode path names in a Windows file system (for example, D:\PythonProjects\SetParameter.project).

This can lead to issues when encoding path names in a Python script.

There are two possible solutions:

- Either encode all backslashes with the `\\` sequence (for example, `project_path = "D:\\PythonProjects\\SetParameter.project"`).
- Or disable the default behavior by prefixing an `r` in front of the opening quotation mark (for example, sequence (for example, `project_path = r"D:\PythonProjects\SetParameter.project"`).

Colons (:)

Loop declarations and conditions end with a colon.

Example:

```
if len(messages) == 0:
        print("--- Build successful ---")
```

EcoStruxure Machine Expert Python API

Detailed Information About Entry Points

Driver	Name (Type)	Description
System	system (ISystem)	basic functionality for integration in the EcoStruxure Machine Expert system
	Severity	ENUM for news priority
	Guid	data type for **G**lobally **u**nique **id**entifier
	PromptResult	ENUM for return values for user request
	MultipleChoiceSelector	delegate type for the selection of multiple choice prompts
	PromptChoiceFilter	
Projects	projects (IScriptProjects)	basic functionality for project management
	ExportReporter	interface for event handling during export
	ImportReporter	interface for event handling during import
	ConflictResolve	ENUM for conflict resolve during import
Online	online (IScriptOnline)	basic functionality for online connection to device
	OnlineChangeOption	ENUM for download types during login to device
	ApplicationState	ENUM for application state
	OperatingState	ENUM for operation state
	ValuesFailedException	exception due to errors detected with online expressions
	TimeoutException	exception on timeout during online operations
DeviceObject	DeviceID	type encapsulation for device identification

For further information, refer to the chapter *Schneider Electric Script Engine Examples* in this document, page 836 or to the chapter Explore Python API (with `dir()` and `inspectapi`), page 822 to explore the API directly.

EcoStruxure Machine Expert Scripting - Python API

Overview

EcoStruxure Machine Expert provides a Python API that you can use in Python scripts.

The Python API consists of two categories:

- Standard Python API and Python modules / packages
- EcoStruxure Machine Expert Python API

The standard Python API (for dealing with strings, arrays, files, and so on) is part of the installation. For detailed help, consult the Internet (for example, https://docs.python.org/3/). You can extend your installation by Python modules and packages available on the Internet.

To automate EcoStruxure Machine Expert, an EcoStruxure Machine Expert Python API is also included in the installation. It is used, for example, to open or close a project, to change the project content or to compile a project and download it to a controller.

The EcoStruxure Machine Expert Python API concept is described in this chapter. Also refer to the script engine examples, page 836. To explore the EcoStruxure Machine Expert Python API, refer to the chapter Explore Python API, page 822.

For details of method / function descriptions or their parameter descriptions, refer to the section *Scripting Engine* in the *Software* part of the online help.

Python API Concept Based on Object-Oriented Programming

To find the API methods suitable to your Python scripts, you must become familiar with the EcoStruxure Machine Expert Python API concept and how the API is integrated into the script engine (script execution). The main concept behind the EcoStruxure Machine Expert Python API is an object-oriented API approach. Object-oriented programming (OOP) is a programming paradigm based on two concepts: objects and code. Objects are data structures that contain data, in the form of fields, also called attributes. Code is available in the form of procedures, also called methods. The procedures of an object can access and modify the data fields of the object with which they are associated. Computer programs that are designed according to OOP consist of objects that interact with one another.

The challenge while writing scripts is to map the detailed EcoStruxure Machine Expert Python API documentation on the right objects to call the procedures.

Predefined Variables and Types

When you run a script or use the REPL (in LogicBuilderShell.exe or **Scripting Immediate** view,) you have a main script scope with predefined variables (and types). You can use them in your script as an entry to the EcoStruxure Machine Expert Python API. To list the available predefined variables in the main script scope, start the LogicBuilderShell.exe and execute `dir()`.

List of predefined variables by running `dir()`:

```
>>> dir()
['AccessRight', 'ApplicationState', 'ArchiveCategories',
'ChannelType', 'Compile
rMessage', 'ConflictResolve', 'ConnectorRole',
'CredentialSourceKind', 'DeviceID
', 'DeviceUserManagementFlags', 'DiagType',
'ExportReporter', 'Guid', 'Implement
ationLanguage', 'ImportReporter', 'MultipleChoiceSelector',
'NativeExportReporte
r', 'NativeImportFilter', 'NativeImportHandler',
'NativeImportResolve', 'NativeI
mportResult', 'ObjectPermissionKind', 'OnlineChangeOption',
'OperatingState', 'P
ermissionState', 'ProjectType', 'PromptChoice',
'PromptChoiceFilter', 'PromptHan
dling', 'PromptResult', 'ResetOption', 'SV_DEV', 'SV_POU',
'Severity', 'TimeoutE
xception', 'ValuesFailedException', 'Version', '__builtins__
',
'__doc__', '__file__', '__name__', 'communication_settings',
'compiler_settings'
, 'etest_test_provider', 'feature_settings_manager',
'librarymanager', 'libraryp
ackage_service', 'new_project', 'online', 'projects',
'system', 'visualization_s
ettings']
```

The predefined entries from this list written in lower case letters are variables which reference to an object that provides methods or fields. This is a kind of global API that works with or without a loaded project.

For example, `system`, or `projects` or `online`.

- `system`: Functionality for integrating EcoStruxure Machine Expert. This object provides the functions that are documented in `ISystemInterface`, such as accessing the **Messages** view from EcoStruxure Machine Expert, or using *ui_present* to verify whether the program is running in `--noUI` mode.
- `projects`: Functionality for project management. This object provides the functions that are documented in `IScriptProjects Interface`, such as loading projects and project archives. Furthermore, this is the entry point to individual projects.
- `online`: Functionality for online access to the controller. You can use the `create_online_application` method to retrieve a specific online object (documented in `IScriptOnlineApplication`) for an application object. This object allows you to log in to a controller, start the application, and read variable values.

For more information, refer to the *Script Engine Plugin API Reference* part of the online help.

The predefined entries from this list starting with an upper case letter are enumerations or types/classes which can be used or instantiated in your Python script. For example, `DeviceID` (class) or `Guid` (class) or `PromptChoice` (enumeration).

Searching and Navigating Within the Tree Structure

A project consists of devices, POUs, DUTs, GVLs, and so on, which are organized in a tree of objects (see, for example, the **Devices tree** in the Logic Builder). This tree of objects (the project tree) is attached to `projects.primary`. In `projects.primary`, the API allows you to search objects by using the `find(…)` method. You can navigate within the project tree, for example, by using the `get_children()` method to get the immediate child objects (for example, the controller devices). On each child object, you can call `get_children()` again to get their immediate child objects, and so on.

NOTE: The methods available on the project tree objects can vary from object to object depending on its type (POU, DUT, GVL, device). It is a best practice to use `inspectapi.dir(…)` to see the list of available API methods on an object.

Examples

Example of using a global API function to open a project :

```
projects.open("MyProject.project")
```

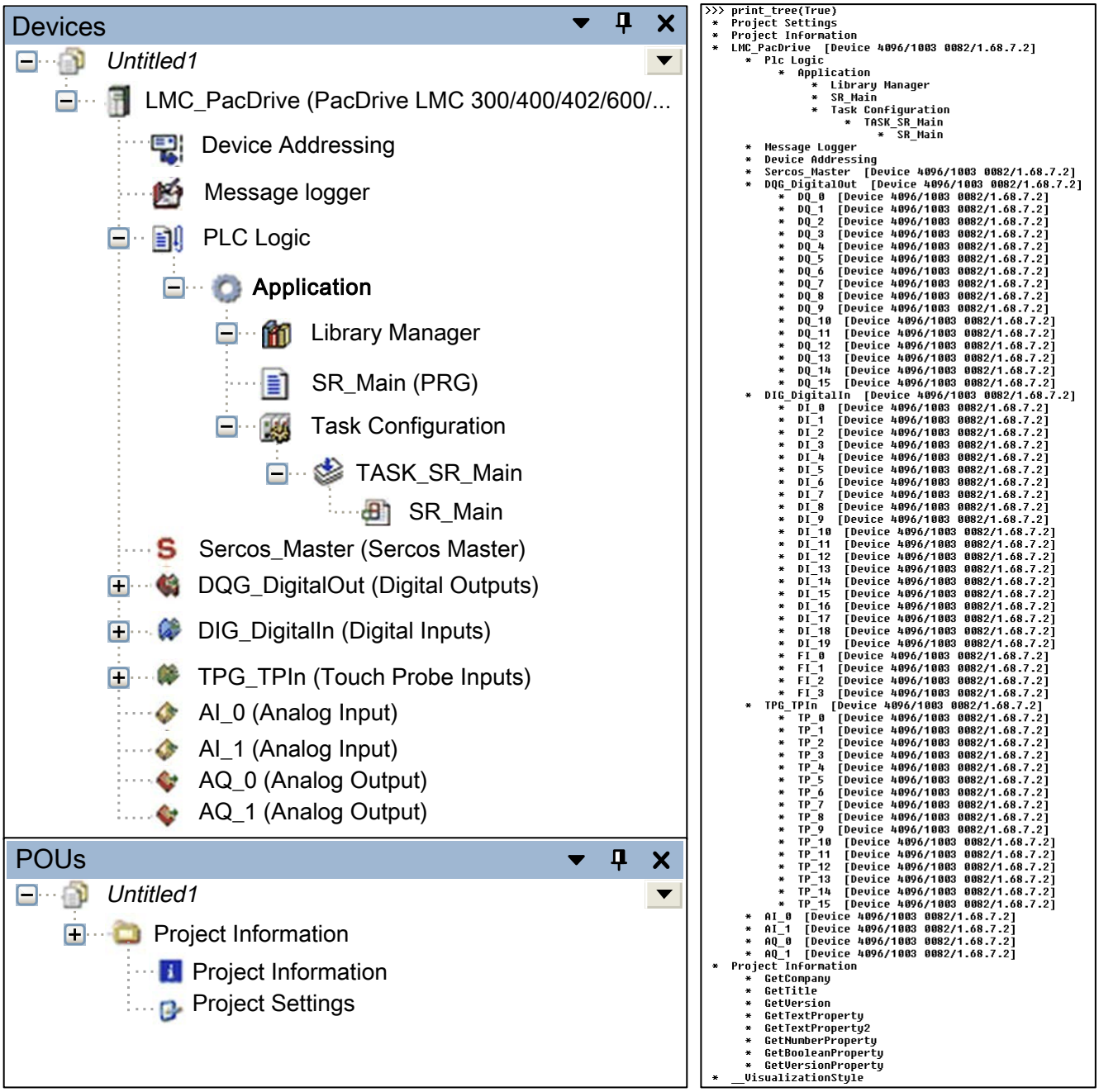
Example of using a global API function to open a file-based storage project (*.fbs):

```
projects.open("MyProject.fbs")
```

Example to find and rename a project tree object (after a project was loaded):

```
myObject = projects.primary.find("SERCOSIII")[0]
myObject.rename("New_SERCOSIII_Name")
```

The figure shows the project tree object in the user interface and the traversed project tree printed by a script:



Example script to traverse the project tree (as printed above):

```
def print_tree_of_obj(treeobj, depth=0, verbose=False):
    name = treeobj.get_name(False)
    if treeobj.is_device:
        deviceid = treeobj.get_device_identification()
        details = ""
        if verbose == True:
```

```
            details = " [Device {0}/{1}/{2}]".format
(deviceid.type, deviceid.id, deviceid.version)
        print("{0} * {1}{2}".format("   "*depth, name,
details))
    else:
        print("{0} * {1}".format("   "*depth, name))

    for child in treeobj.get_children(False):
        print_tree_of_obj(child, depth+1, verbose)
def print_tree_of_project(pro, verbose=False):
    if pro == None:
        print("No project open.")
    else:
        for obj in pro.get_children():
            print_tree_of_obj(obj, 0, verbose)
def print_tree(verbose=False):
    if projects.primary != None:
        print_tree_of_project(projects.primary, verbose)
    else:
        print("No project open.")
print_tree(True)
```

Calling Scripts Via Toolbar Icons

Overview

You can configure up to 32 icons that will be displayed in a toolbar to access Python script files with one click.

Components required:

- An .ico file defining the icon.
- A .py file storing the Python script.
- A *config.json* file specifying the call information outline for each icon. A maximum of 16 icons can be configured per file.

To configure more than 16 icons, create two *config.json* files and store each file at one of the following locations:

- *C:\Program Files\Schneider Electric\EcoStruxure Machine Expert\Script Commands*
- *C:\Program Files\Schneider Electric\EcoStruxure Machine Expert\V2.x \LogicBuilder\Script Commands*

When you start your program, the script calls and the related icons are added to the dialog box **Tools > Customize** in the tab **Command Icons**, category **ScriptEngine Commands**. You can add them to a new or to an existing toolbar in the **Toolbars** tab of the **Tools > Customize** dialog box.

Elements of the *config.json* Configuration File

Each *config.json* configuration file contains the following required and optional elements:

Element	Required	Description
"Name"	Yes	This text is displayed as a tooltip for the icon in the toolbar. Example: `"Name": "Pause"`
"Desc"	No	This element is not used.
"Icon"	Yes	This element contains the name of the image file (in .ico format) if the image file is located in the same folder (*Script Commands*) as the *config.json* file. Example: `"Icon": "pause.ico"` **NOTE:** To help prevent cyberattacks, a path is not supported. To place the image file into the same folder as the *config.json* file, (Windows) administrator rights are required on your computer.
"Path"	Yes	This element contains the name of the Python script file in .py format if the script file is located in the same folder (*Script Commands*) as the *config.json* file. Example: `"Path": "stop.py"` **NOTE:** To help prevent cyberattacks, a path is not supported. To place the script into the same folder as the *config.json* file , (Windows) administrator rights are required on your computer.
"Params"	No	Use this element only if the script requires parameters. Example: `"Params": [ "file1", "file2" ]`

Example of a *config.json* File

A *config.json* file can have the following content:

```
[
    {
        "Name": "Start Process1",
        "Desc": "Processing1",
        "Icon": "process1.ico",
        "Path": "proc1.py",
    },
    {
        "Name": "Start Process2",
        "Desc": "Processing2",
        "Icon": "process2.ico",
        "Path": "proc2.py",
        "Params":
                [
                        "exit"
                ]
        }
]
```

The folder Script Commands thus contains the following files:

- *config.json*
- *process1.ico*
- *proc1.py*
- *process2.ico*
- *proc2.py*

The **ScriptEngine Commands** list of the **Tools > Customize** dialog box contains the following entries that you can add to a toolbar:

- Start Process1
- Start Process2

Click an icon in the toolbar to execute the respective script file.

Schneider Electric Script Engine Examples

New Projects

Overview

This chapter shows how to create a new project with the Schneider Electric Script Engine.

Even though there are 4 different project types available, the code for creating them is identical. In these 4 cases, the *create_project* method of the global object *new_project* is called.

This instantiates 2 data containers that consist of the following elements:

- common project settings
- controller settings

Common Project Settings

The common project settings consist of the following elements:

- machine name
- author
- customer
- description
- picture (the path to an image)
- project name (the name of the project including the file extension *.project*)
- project type

The project type property allows you to select one of the 3 project types:

Property Value	Description
`StandardProject`	standard project
`Library`	library
`EmptyProject`	empty project

Controller Settings

The controller settings consist of the following elements

Element	Description
type	type of the controller, can be retrieved from the device identification of the controller
id	ID of the controller, can be retrieved from the device identification of the controller
version	firmware version
device_name	name of the device that you can assign
implementation_language	6 implementation languages are available

Creating a Standard Project

To create a standard project, the *create_project* method is called. It consists of the data containers *CommonProjectSettings* and *ControllerSettings*.

```
# Clean up any open project
if projects.primary:
    projects.primary.close()

# Create a new instance of the common project settings
object
common_settings_DTO = new_project.create_common_project_
settings()
common_settings_DTO.machine_name = "Machine Name"
common_settings_DTO.author = "Author"
common_settings_DTO.customer = "Customer"
common_settings_DTO.description = "Description"
common_settings_DTO.picture = None
common_settings_DTO.project_name = "Example.project"
common_settings_DTO.project_path = r"D:\PythonProjects"
common_settings_DTO.project_type = ProjectType.
StandardProject

# Create a new instance of the controller settings object
controller_settings_DTO = new_project.create_controller_
settings()
controller_settings_DTO.type = 4096
controller_settings_DTO.id = "1003 0082"
controller_settings_DTO.version = "1.53.5.4"
controller_settings_DTO.device_name = "LMC_PacDrive"
controller_settings_DTO.implementation_language =
ImplementationLanguage.structured_text

# Create new standard project
new_project.create_project(common_settings_DTO, controller_
settings_DTO)

# Save the project
projects.primary.save()
```

Device Parameters

Overview

To change a parameter, the parameter ID and the `ParameterSet` are required.

In order to find the required device and list the respective parameters, use the find method that finds objects by a given name or path in the project.

Script Engine Example

```
from __future__ import print_function

# Example-Device Lexium62
moduleName = 'DRV_Lexium62'

# find module/device in Project (you also can do this
generic e.g. with get_children()...)
module = projects.primary.find(moduleName, True)[0]

# get all possible connectors for this device
connectors = module.connectors
print("Number of Connectors: ", connectors.Count)
print(" - ")

cnt = 1
for c in connectors:
        print("Connector " +str(cnt) +" for "+moduleName +":
")
        params = c.host_parameters
        count = params.Count
        print("Number of available Parameters for this
connector: ", count)

        if count > 0:
            for p in params:
                    print(p.visible_name + " - ID: %s " %
p.id + " - Value: %s" % p.value)

            # example to get the value of a parameter by id:
            print(" - ")
            print(" - Example for a single Parameter
(ObjectType) - ")
            objectType = params.by_id(268435457)
            print("ObjectType: ", objectType.value)

        cnt = cnt+1

        print
(-
"----------------------------------------------------------------------")
```

Compiler Version

Overview

With the compiler version extension, you can display the mapped compiler versions and you can set a new compiler version by executing the script.

Script Engine Example

```
# Enable the new python 3 print syntax
from __future__ import print_function

# The path to the project
project_path = r"D:\PythonProjects\GetCompilerVersion.
project"

# Clean up any open Project:
if projects.primary:
    projects.primary.close()
```

```
# Load the project
proj = projects.open(project_path);

print("All compiler versions")
# Get all compiler versions (filtered)
compiler_versions = compiler_settings.get_all_compiler_
versions()

# Print all compiler versions (filtered)
for version in compiler_versions:
    print (" - OEM mapped version: " + version)

# Get active compiler version
compiler_version = compiler_settings.active_compiler_version
print("Current compiler version:" + compiler_version)

# Set new compiler version
compiler_settings.active_compiler_version = "4.2.0.0"
print("New compiler version: " +
compiler_settings.active_compiler_version)

# Save project
projects.primary.save()
```

Update Project

Overview

With the update project extension, you can update your project. This is the same function as provided by the **Project Update** dialog box that is displayed when you open the project in the graphical user interface and click **OK**.

Script Engine Example

```
# Enable the new python 3 print syntax
from __future__ import print_function

# The path to the project
project_path = r"D:\PythonProjects\Example.project"

# Clean up any open project:
if projects.primary:
    projects.primary.close()

# Load the project
proj = projects.open(project_path)
# Set the project as primary project
proj = projects.primary

# Update the project (devices, libraries, compiler version,
visualization style)
proj.update_project()
```

Update Libraries

Overview

With the update libraries extension, you can update the libraries of the project automatically. This is the same function as provided by the **Libraries > Automatic version mapping (all libraries)** command in the graphical user interface.

Script Engine Example

```
# Enable the new python 3 print syntax
from __future__ import print_function

# The path to the project
project_path = r"D:\PythonProjects\Example.project"

# Clean up any open project:
if projects.primary:
    projects.primary.close()

# Load the project
proj = projects.open(project_path);

# Search for the library manager objects in the project
lib_managers = [i for i in proj.get_children(True) if i.is_
libman]

# Make the auto mapping for each library manager found
for lib_manager in lib_managers:
    lib_manager.make_auto_mapping()
```

Clean and Build Application

Overview

With the clean and build application extension, you can clean a project or build a new project.

Script Engine Example

```
# Enable the new python 3 print syntax
from __future__ import print_function

# The path to the project
project_path = r"D:\PythonProjects\Example.project"

# Clean up any open project:
if projects.primary:
    projects.primary.close()

# Load the project
proj = projects.open(project_path);

# Fetch the active application.
app = proj.active_application

# Clean application
app.clean()

# Compile application and store compiler messages in a list
app.build()
for msg_cat in system.get_message_categories():
    if system.get_message_category_description(msg_cat) == "Build":
        messages = system.get_message_objects(msg_cat, Severity.Warning |
Severity.Error | Severity.FatalError)
        break
# If length of messages list is 0, the build was successful
    print("--- Build successful ---")

# Otherwise print results
else:
```

```
for msg in messages:
    if hasattr(msg, 'object'):
        texts = [str(msg.severity), msg.text, msg.object.get_name()]
    else:
        texts = [str(msg.severity), msg.text]

    print("; ".join(texts))
```

Communication Settings

Overview

This example shows how to load or set the IP address of a controller of your choice by executing the script.

Script Engine Example

```
from __future__ import print_function

def main():
    if not projects.primary:
        system.ui.error("No active project.")
        return
    project = projects.primary

    #find the controller by the object name where address
should be set and read.
    controller = project.find('LMC_PacDrive', True)[0]

    #reboot the controller
    controller.set_communication_address('192.168.2.25')

    #read address back and show it.
    print('get_communication_address := ' + controller.get_
communication_address())


main()
```

Start ETEST

Overview

In software programming, the module ETEST is used to verify the correct functionality of individual methods or algorithms of a program. ETEST is a white-box testing method, meaning that the source code to be verified is known when designing test cases and determining the expected results.

The procedure typically consists of 3 steps:

- initializing the initial state
- executing the operation that is to be tested
- comparing the actual results with the expected results

The ETEST script driver performs the following tasks:

- displaying test elements
- searching particular tests
- executing particular tests

Refer to the following example.

Script Engine Example

```
# Enable the new python 3 print syntax
from __future__ import print_function

# The path to the project
project_path = r"D:\PythonProjects\Example.project"

# Clean up any open project:
if projects.primary:
    projects.primary.close()

# Load the project
proj = projects.open(project_path)

# Fetch the active application.
app = proj.active_application

# Create the online application for it.
onlineapp = online.create_online_application(app)

# Log in to the device.
onlineapp.login(OnlineChangeOption.Try, True)

# Start the application, if necessary.
if not onlineapp.application_state == ApplicationState.run:
    onlineapp.start()

# Let the app do its work for some time...
system.delay(1000)

# This function runs the specified test/test series
def func_run_test(test_name):
    print(test_name)
    etest_test_provider.run_test(test_name)
    while etest_test_provider.is_test_running:
        system.delay(1000)
    return

# Get all test series from the project
test_series = etest_test_provider.get_all_testseries()

# Run all test series
for test_object in test_series:
    func_run_test(test_object)

# Get all test elements from test series "TS_MySeries"
test_elements = etest_test_provider.get_all_testelements
("TS_MySeries")

if test_elements == None:
    print ("No test series 'TS_MySeries' found")
else:
    # Print all testelements
    for test_element in test_elements:
        print ("Found test element in Series TS_MySeries: "
+ test_element.Name)

        # Run only "TS_Crank" test object
        func_run_test(test_element.Name)

onlineapp.logout()

projects.primary.close()
```

Reset Diagnostic Messages

Overview

Once logged in to the application, you can reset the diagnostic messages of the controller. This is an extension method of the controller object.

The following example shows how to reset diagnostic messages. You have to get the primary project and log in to the application, as shown in the other examples (building an application, page 840).

Script Engine Example

```
# get the project instance and log in to the application

# find the controller which messages shall be reset
controller = project.find("LMC", True)[0]

# "reset diagnosis messages on controller"
controller.reset_diagnosis_messages()
```

Reboot the Controller

Overview

Once there is an instance of the controller object in the script, you can reboot the controller by using a method on that object.

Script Engine Example

```
from __future__ import print_function

def perform_application_login(project):
    app = project.active_application
    onlineapp = online.create_online_application(app)
    onlineapp.login(OnlineChangeOption.Try, True)

def main():
    if not projects.primary:
        system.ui.error("No active project.")
        return

    perform_application_login(projects.primary)

    #find the controller named 'LMC_PacDrive' which shall be
rebooted
    controller = projects.primary.find('LMC_PacDrive', True)
[0]

    #reboot the controller
    controller.reboot_plc()

    system.ui.info("Download or OnlineChange complete")


main()
```

Convert Device

Overview

Converting devices within the project can become a complex procedure. This API simplifies the conversion process and helps to avoid errors.

Using the *DeviceID* Object

The conversion API uses the *DeviceID* object which identifies a device or a device module by a specific version. The *DeviceID* is created as follows:

<device type> <device model> <device version> <module name>

Element	Example	Description
device type	`4096`	identifies a controller
device model	`1003 0082` or `1003 009D`	for LMCx00C or LMCx01C, respectively
device version	`1.50.0.4`	firmware version of the controller
module name	`LXM52`	target device module

The conversion API accepts the `DeviceID` as object instance or as single parameter. This allows using a `DeviceID` containing all elements mentioned in the above table or passing each element as a single parameter.

The following example shows how to create a *DeviceID* for an LMCx00C controller with firmware version 1.50.0.4:

```
Lmcx00c = DeviceID(4096, "1003 0082", "1.50.0.4")
```

Testing Whether a Device Can be Converted

The following script allows you to verify whether a conversion to a given version is possible before converting a device.

```
from __future__ import print_function

def main():
    # Set the project as primary project
    proj = projects.primary
    controller = proj.find('LMC_PacDrive', True)[0]
    drives = controller.find('DRV_Lexium62', True)

    if len(drives) == 0:
        print("Expected drive object not found")
        return

    drive = drives[0]

    # test if controller can be converted using DeviceID
    x01c = DeviceID(4096, "1003 009D", "1.53.5.4")
    if controller.can_convert(x01c):
        print("Conversion to LMCx01C possible")

    # test if drive can be converted using Parameters and
module id
    if drive.can_convert(4096, "1003 0082", "1.53.5.4",
"LXM52"):
        print("Conversion to LXM52 possible")


if not projects.primary:
    print("No project open.")
```

```
else:
    main()
```

Getting Alternative Conversion Targets

The API provides a call that retrieves the possible conversion targets for a certain device. It returns the *DeviceID* for each target.

```
from __future__ import print_function

#help function to print the delivered device ids
def deviceid_to_string(devId):
    mystr = "ID: {0.id} Type: {0.type} Version: {0.
version}".format(devId)

    if hasattr(devId, 'module_id') and devId.module_id is
not None:
        mystr += " ModuleID: {0.module_id}".format(devId)
        return mystr


def main():

    # Set the project as primary project
    proj = projects.primary

    controller = proj.find('LMC_PacDrive', True)[0]
    alternativeControllers = controller.get_alternative_
devices()

    print("ALTERNATIVE DEVICES FOR LMC")
    for id in alternativeControllers:
        print(deviceid_to_string(id))

    drive = proj.find('DRV_Lexium62', True)[0]
    alternativeDrives = drive.get_alternative_devices()

    print("ALTERNATIVE DEVICES FOR DRIVE")
    for id in alternativeDrives:
        print(deviceid_to_string(id))

    print("Test complete. Please check the script output
window")



if not projects.primary:
    print("No project open.")
else:
    main()
```

Converting the Device

The process of converting the device is straightforward because the only required action is calling the conversion method.

```
from __future__ import print_function

def main():
    proj = projects.primary
    controller = proj.find('LMC_PacDrive', True)[0]
    drive = proj.find('DRV_Lexium62', True)[0]

    # converting the controller
    controller.convert(4096, "1003 009D", "1.53.5.4")
    # converting the drive
```

```
    drive.convert(DeviceID(4096, "1003 0082", "1.53.5.4"),
"LXM52")


if not projects.primary:
    print("No project open.")
else:
    main()
```

Comparing Projects

Overview

There are several use cases where it is useful to have a script automatically comparing the contents of two projects. The Python project comparison function allows you to compare two projects. As a result it provides the information if the projects are different.

Script Engine Example

```
from __future__ import print_function

def main():
    proj = projects.primary

    # compare the Primary Project to another Project on disk
    proj2 = projects.open(r"D:\PythonProjects\CompTest_Right.project", primary=
False)
    diff = proj.compare_to(proj2)
    write_diff(diff, r"D:\PythonProjects\Diff1.txt")

    # compare, but ignore whitespaces, comments and properties
    diff = proj.compare_to(proj2, ComparisonFlags.IGNORE_WHITESPACE |
ComparisonFlags.IGNORE_COMMENTS | ComparisonFlags.IGNORE_PROPERTIES)
    write_diff(diff, r"D:\PythonProjects\Diff2.txt")


def write_diff(differences, filename):
    content = []
    sep = "; "
    objects = differences.get_changed_objects()
    for obj in objects:
        if obj.differences != ObjectDifferences.EQUAL:
            right_obj = "no_object"
            left_obj = "no_object"
            if obj.right_object is not None:
                right_obj = obj.right_object.get_name()
            if obj.left_object is not None:
                left_obj = obj.left_object.get_name()
            content.append(sep.join([left_obj, right_obj, str(obj.differences)]) +
"\n")

    if len(content) != 0:
        with open(filename, 'wb') as f:
            f.writelines(content)
    else:
        print("Projects are equal.")


if not projects.primary:
    print("No project open.")
else:
    main()
```

Advanced Library Management Functions

Overview

The Logic Builder offers advanced functions for managing libraries, the so-called forward compatible libraries (see Functions and Libraries User Guide). They provide a convenient way to manage references and dependencies among libraries.

This functionality is also available via scripts and can be used on the **Library Manager** of the entire project or on a single application within the project. The following script shows how to check the libraries for forward compatibility and valid references. It automatically maps the references and explicitly sets library versions.

Script Engine Example

```
proj = projects.primary
app = proj.active_application

libmgr = app.get_library_manager()
print("# Checking all libraries:")
for lib in libmgr.get_libraries():
    print("- " + lib + " Is Forward Compatible Library? " +
str(libmgr.is_library_forward_compatible(lib)))

if not libmgr.is_current_mapping_valid():
    for lib in libmgr.get_invalid_library_mappings():
        print("Library reference cannot be satisfied for: "
+ lib)
    print("Trying to auto-map libraries to valid versions")
    libmgr.make_auto_mapping()
else:
    print("All mappings valid")

# set version using individual parameters
libmgr.set_new_library_version("PD_GlobalDiagnostics",
"Schneider Electric", "1.0.1.0")

# set version using the library full name
libmgr.set_new_library_version("PD_AxisModule, 1.1.6.0
(Schneider Electric)", "1.2.4.0")

# set version to Legacy
libmgr.set_new_library_version("PD_Template", "Schneider
Electric", None)
```

Accessing POUs

Overview

The following examples show how to print and manipulate the code of a POU. They are only available for textual programming languages.

Script Engine Example

```
if not projects.primary:
    print('No primary project set')

proj = projects.primary
pou = proj.find('SR_Main', True)[0]
```

```
# read and print the declaration of the program
decl = pou.textual_declaration
print(decl.text)

# read and print the implementation of the program
code = pou.textual_implementation
print(code.text)

decl_text = "PROGRAM SR_Main\n" + \
            "VAR\n" + \
            "  iTest: INT;\n" + \
            "END_VAR";

code_text = "iTest := iTest + 1;"

# write new code to the declaration and implementation
decl.replace(decl_text)
code.replace(code_text)
```

Script Engine Example for Obsolete set_interface_text() API

```
if not projects.primary:
    print("No primary project set")

proj = projects.primary
pou = proj.find('SR_Main', True)[0]

# read and print the declaration of the program
decl = pou.get_interface_text()
print(decl)

# read and print the implementation of the program
code = pou.get_implementation_text()
print(code)

decl = "PROGRAM SR_Main\n" + \
       "VAR\n" + \
       "  iTest: INT;\n" + \
       "END_VAR";
code = "iTest := iTest + 1;"

# write new code to the declaration and implementation
pou.set_interface_text(decl)
pou.set_implementation_text(code)
```

CODESYS Script Engine Examples

Overview

This chapter provides examples of frequently used members of the CODESYS Script Engine. For a complete description of the members of each namespace, refer to the CODESYS API description.

Project

Overview

Since the examples for this namespace are relatively short and self-explanatory, their meaning is not explained in detail. Complete examples are provided, where appropriate.

New Project

This method creates a new project.

It consists of 2 parameters:

- a string specifying the location where the project will be stored
- a boolean parameter: If TRUE, the project will be the new primary project. This parameter is optional, the default value is TRUE.

The method returns the *IProject* instance (refer to the specification in the document *Automation Platform SDK*) which can be used for further steps.

```
import os

try:
    # Clean up any open project
    if projects.primary:
        projects.primary.close()

    # Define the new file name for the project
    project_name = "Example.project"

    # Define the new path where the new project should be
stored
    project_path = r"C:\Python"

    # Create the new project
    proj = projects.create(os.path.join(project_path,
project_name), True)

    # Save the project to the specified path
    proj.save()
except Exception as exception:
    print("Error: " + str(exception))
    if not system.trace:
       print("Please turn on the 'Script Tracing' function
to get detailed information about the script execution.")
```

Load Project

This method loads a project. Any open projects will not be closed.

The first parameter specifies the path of the project that will be loaded.

```
import os

try:
    # Clean up any open project
    if projects.primary:
        projects.primary.close()

    # Define the file name for the project
    project_name = "Example.project"

    # Define the path where the project is stored
    project_path = r"C:\Python"

    # Load the existing project
```

```
    proj = projects.open(os.path.join(project_path, project_
name))
except Exception as exception:
    print("Error: " + str(exception))
    if not system.trace:
       print("Please turn on the 'Script Tracing' function
to get detailed information about the script execution.")
```

Save Project

This method saves the project at its physical location.

```
# Save project
projects.primary.save()
```

Save Archive

This method saves the project as an archive. The additional categories which are selected by default are included, but no extra files.

The first parameter specifies the path where the archive will be saved.

```
import os

# Define the new file name for the archive
archive_name = "Example.archive"

# Define the new path where the archive should be stored
archive_path = r"C:\Python"

# Save archive with the default values
projects.primary.save_archive(os.path.join(archive_path,
archive_name))
```

Close Project

This method closes the project. If there are unsaved changes in this project, these changes will be discarded.

```
# Clean up any open project:
if projects.primary:
    projects.primary.close()
```

Find Objects

This method finds objects matching the given name.

It consists of 2 parameters:

- The first parameter is the name of the object that is searched.
- The second parameter specifies whether a recursive search is performed. This parameter is optional, the default value is FALSE. This method returns a collection of objects.

```
# Search for
result_list = projects.primary.find('MyController', True)

for result in result_list:
    print("Object " + result.get_name() + " found with Guid
" + str(result.guid))
```

Names are not unique in the tree. This has the effect that several objects can be found. The search is against the nonlocalized name.

Native Import

This method imports the specified files in the native XML format in the top level of this project.

```
import os

# Specify the project file name
project_name = "NativeImport.project"

# Define the path where the project should be/is stored
project_path = r"C:\Python"

# Define the path where the exported objects are be stored
object_path = os.path.join(project_path, "Objects")

# Create the import reporter
class Handler(NativeImportHandler):
    def conflict(self, name, obj, guid):
        print("Object already exists: " + name)
        return NativeImportResolve.skip

    def progress(self, name, obj, exception):
        print("in progess: " + name)

    def skipped(self, list):
        return

def import_filter(name, guid, type, path):
    # Workaround, skip the project settings object because
we cant import it
    if(type == "_3S.CoDeSys.Engine.WorkspaceObject"):
        return False
    return True

try:
    # Clean up any open project
    if projects.primary:
        projects.primary.close()

    # Create the new project
    project_reference = projects.create(os.path.join
(project_path, project_name), True)

    files = os.listdir(object_path)

    # Create the importer instance.
    handler = Handler()

    for file in files:
        file_path = os.path.join(object_path, file)
        project_reference.import_native(file_path, import_
filter, handler)

    project_reference.save()

except Exception as exception:
    print("Error: " + str(exception))
    if not system.trace:
       print("Please turn on the 'Script Tracing' function
to get detailed information about the script execution.")
```

PLCOpenXML Import

This method imports the contents of the specified PLCopenXML file into the top level of the project.

```
import os
```

```
# Specify the project file name
project_name = "PLCOpenXMLImport.project"

# Define the path where the project should be/is
storedproject_path = r"C:\Python"

# Define the file where the exported object is stored
file_name = os.path.join(project_path, r"Objects
\MyController.xml")

# Create the import reporter
class Reporter(ImportReporter):
    def error(self, message):
        system.write_message(Severity.Error, message)
    def warning(self, message):
        system.write_message(Severity.Warning, message)
    def resolve_conflict(self, obj):
        return ConflictResolve.Copy
    def added(self, obj):
        print("added: ", obj)
    def replaced(self, obj):
        print("replaced: ", obj)
    def skipped(self, obj):
        print("skipped: ", obj)
    @property
    def aborting(self):
        return False

try:
    # Clean up any open project
    if projects.primary:
        projects.primary.close()

    # Create the reporter instance
    reporter = Reporter()

    # Create the new project
    project_reference = projects.create(os.path.join
(project_path, project_name), True)

    # Import the data into the project
    project_reference.import_xml(reporter, file_name)

    # Save the project to the specified path
    project_reference.save()
except Exception as exception:
    print("Error: " + str(exception))
    if not system.trace:
       print("Please turn on the 'Script Tracing' function
to get detailed information about the script execution.")
```

Native Export

This method exports the given objects in native format into a string, or a file at the given path. The non-exportable objects are detected as an error, but the export continues.

```
import sys,io,os

# Specify the project file name
project_name = "Example.project"

# Define the path where the project should be/is stored
project_path = r"C:\Python"

# Define the path where the exported objects should be
stored
```

```
object_path = os.path.join(project_path, "Objects")

def collect_objects(project_reference):
    # List that stores all POU nodes
    project_objects = []

    # Collect all the leaf nodes.
    for node in project_reference.get_children(True):
        project_objects.append(node)

    for i in project_objects:
        print("Found: ", i.type, i.guid, i.get_name())
    return project_objects

def export_objects(collected_objects, project_reference):
    if not os.path.exists(object_path):
        os.makedirs(object_path)

    # Export the files.
    for candidate in collected_objects:
        # Create a list of objects to export:
        # The object itself
        objects = [candidate]

        # And sub-objects (POUs can have actions,
properties, ...)
        objects.extend(candidate.get_children(True))

        # And the parent folders.
        parent = candidate.parent
        while ((not parent.is_root) and parent.is_folder):
            objects.append(parent)
            parent = parent.parent

        # Create a unique file name
        filename = os.path.join(object_path, "%s__%s.export"
% (candidate.get_name(), candidate.guid))

        # Print some user information
        print("Exporting " + str(len(objects)) + " objects
to: " + filename)

        # And actually export the project
        project_reference.export_native(objects, filename)

try:
    # Clean up any open project
    if projects.primary:
        projects.primary.close()

    # Open a project first
    project_reference = projects.open(os.path.join(project_
path, project_name))

    # Collect the objects
    collected_objects = collect_objects(project_reference)

    export_objects(collected_objects, project_reference)
except Exception as exception:
    print("Error: " + str(exception))
    if not system.trace:
       print("Please turn on the 'Script Tracing' function
to get detailed information about the script execution.")
```

PLCOpenXML Export

This method exports the given objects in PLCopenXML format into a string, or a file at the given path. The non-exportable objects are detected as an error, but the export continues.

```
import os

# Specify the object which should be exported
object_name = "MyController"

# Define the path where the exported objects should be
stored
object_path = r"C:\Python\Objects"

# Define the printing function
def print_tree(treeobj, depth=0):
    name = treeobj.get_name(False)
    if treeobj.is_device:
        deviceid = treeobj.get_device_identification()
        print("{0} - {1} {2}".format(" "*depth, name,
deviceid))

    for child in treeobj.get_children(False):
        print_tree(child, depth+1)

# Create the export reporter
class Reporter(ExportReporter):
    def error(self, message):
        system.write_message(Severity.Error, message)
    def warning(self, message):
        system.write_message(Severity.Warning, message)
    def nonexportable(self, message):
        print(message)
    @property
    def aborting(self):
        return False

try:
    # Get the project reference of the currently opened
project
    project_reference = projects.primary

    # Get a reporter instance
    reporter = Reporter()

    # Print all devices in the project
    for obj in project_reference.get_children():
        print_tree(obj)

    # Finds the object in the project, and return the first
result
    device = project_reference.find(object_name, True)

    if device != None:
        filename = os.path.join(object_path, device[0].get_
name() + ".xml")

        # Exports the object to the hard drive
        project_reference.export_xml(reporter, device,
filename, True, True)

except Exception as exception:
    print("Error: " + str(exception))
    if not system.trace:
       print("Please turn on the 'Script Tracing' function
to get detailed information about the script execution.")
```

Online Application

Overview

NOTE: Some of the online application commands can temporarily change the active application.

This interface is exported to Python, and thus complies to Python naming standards.

Create Online Application

This method creates an online application.

```
# Create online application
online_application = online.create_online_application()
```

Prepared Values

This example shows how to write prepared values.

```
try:
    # Trying to create online application
    online_application = online.create_online_application()

    print("app:" + str(online_application.application_state)
+ "op:" + online_application.operation_state.ToString("f"))

    for expression in online_application.get_prepared_
expressions():
        online_application.set_prepared_value(expression,
'')
        print "%s: '%s' '%s'" % (expression, online_
application.read_value(expression), online_application.get_
prepared_value(expression))

    for expression in online_application.get_forced_
expressions():
        online_application.set_unforce_value(expression)
        print "%s: '%s' '%s'" % (expression, online_
application.read_value(expression), online_application.get_
prepared_value(expression))

    assert len(online_application.get_prepared_expressions
()) == 0, "Still some prepared values remain."
    assert len(online_application.get_forced_expressions())
== 0, "Still some prepared values remain."

    # Preparing a value and forcing it
    online_application.set_prepared_value("POU.testoutput",
"123")
    online_application.force_prepared_values()

    # Preparing a value and writing it    online_
application.set_prepared_value("POU.testint", "INT#1147")
    online_application.write_prepared_values()

except Exception as exception:
    print("Error: " + str(exception))
    if not system.trace:
       print("Please turn on the 'Script Tracing' function
to get detailed information about the script execution.")
```

Perform Application Login

This method performs the application login. If the application was logged in before, it will be logged out and a fresh login will be performed.

It consists of 2 parameters:

- The first parameter is the change option.
- The second parameter will delete previous applications, if set to `True`.

```
# Get the project reference
project_reference = projects.primary

# Fetch the active application
active_application = project_reference.active_application

# Create the online application for it
online_application = online.create_online_application
(active_application)

# Log in to the device.
online_application.login(OnlineChangeOption.Try, True)
```

Logout Application

This method logs out the application. If the application is not logged in, nothing happens.

```
# Get the project reference
project_reference = projects.primary

# Fetch the active application
active_application = project_reference.active_application

# Create the online application for it
online_application = online.create_online_application
(active_application)

# Log in to the device.
online_application.logout()
```

Start Application

This method starts the application.

```
# Get the project reference
project_reference = projects.primary

# Fetch the active application
active_application = project_reference.active_application

# Create the online application for it
online_application = online.create_online_application
(active_application)

# Log in to the device.
online_application.login(OnlineChangeOption.Try, True)

# Start the application, if necessary
if not online_application.application_state ==
ApplicationState.run:
    online_application.start()

    # Let the app do its work for some time
    system.delay(1000)
```

Stop Application

This method stops the application.

```
# Get the project reference
project_reference = projects.primary

# Fetch the active application
active_application = project_reference.active_application

# Create the online application for it
online_application = online.create_online_application
(active_application)

# Log in to the device.
online_application.login(OnlineChangeOption.Try, True)

# Stop the application, if necessary
if online_application.application_state == ApplicationState.
run:
    online_application.stop()
```

Objects

Find

This method finds objects matching the given name.

It consists of 2 parameters:

- The first parameter is the name of the object that is searched.
- The second parameter specifies whether a recursive search is performed. This parameter is optional, the default value is FALSE. This method returns a collection of objects.

```
# Search for the specified object
results = projects.primary.find('MyController', True)

for result in results:
    print("Object " + result.get_name() + " found with Guid
" + str(result.guid))
```

Remove

This method removes the object.

```
# Finds the MyController object in the project
results = projects.primary.find("MyController", True)

for result in results:
    # Removes all objects with the name MyController
    result.remove()
```

Rename

This method renames the object to the new name.

```
# Finds the MyController object in the project
results = projects.primary.find("MyController", True)

for result in results:
    # Rename all MyController objects to MyController_2
    result.rename("MyController_2")
```

Import/Export

Refer to the description of imports/exports for projects, page 851. The only difference is that the object is called for import / export, instead of the project.

Devices

Overview

This chapter describes methods for manipulating device objects.

Add

This method adds the specified device.

It consists of 3 parameters:

- A string specifying the name of the device.
- *DeviceID* specifying the ID of the device. For the *DeviceID* value, refer to the device description of the device you want to add.
- A string specifying the module ID.

```
# We want to add a TM5C12D6T6L Module to the TM5_Manager
# Search for the TM5_Manager. The new Module will be
inserted below this object.
parent_device = projects.primary.find('TM5_Manager', True)
[0]

if not(parent_device == None):
    parent_device.add("Module_TM5C12D6T6L", DeviceID(51063,
"101a 0363", "3.1.2.2"))
```

Disable

This method marks this device as disabled during download.

```
# Finds the MyController object in the project
device = projects.primary.find("MyController", True)[0]

if not(device == None):
    device.disable()
```

Enable

This method marks this device as enabled during download.

```
# Finds the MyController object in the project
device = projects.primary.find("MyController", True)[0]

if not(device == None):
    device.enable()
```

Get Address

This method gets the address of the device. It returns a string.

```
# Finds the MyController object in the project
device = projects.primary.find("MyController", True)[0]

if not(device == None):
    print("Address: " + device.get_address())
```

Get Device Identification

This method gets the device identification.

```
# Finds the MyController object in the project
device = projects.primary.find("MyController", True)[0]

if not(device == None):
    print(str(device.get_device_identification()))
```

Get Gateway

This method returns the *GUID* of the gateway.

```
# Finds the MyController object in the project
device = projects.primary.find("MyController", True)[0]

if not(device == None):
    print("Gateway: " + str(device.get_gateway()))
```

Insert

This method inserts the specified device at the specified index.

It consists of 4 parameters:

- A string specifying the name of the device.
- *Int32* specifying the index where to insert the device.
- *DeviceID* specifying the ID of the device. For the *DeviceID* value, refer to the device description of the device you want to add.
- A string specifying the module ID.

```
# Finds the MyController object in the project
parent_device = projects.primary.find("TM5_Manager", True)
[0]

if not(parent_device == None):
    # We need the amount of children to add the new device
at the end
    child_count = len(parent_device.get_children())

    # Use a unique name for the inserted device
    parent_device.insert("Module_TM5C12D6T6L_" + str(child_
count), child_count, DeviceID(51063, "101a 0363",
"3.1.2.2"))
```

Set Gateway and Address

This method sets the gateway and the address. If you pass the empty *GUID* and an empty address, the gateway address will be cleared.

```
# Finds the MyController object in the project
device = projects.primary.find("MyController", True)[0]

if not(device == None):
    device.set_gateway_and_address(gateway, address)
```

Set Simulation Mode

This method sets the simulation mode. If set to *True*, simulation is enabled.

```
# Finds the MyController object in the project
device = projects.primary.find("MyController", True)[0]

if not(device == None):
```

```
    device.set_simulation_mode (True)
```

Update

This method updates the specified device.

```
# This is a more generic approach to find and update a
device by its device description
import os

def find_device_repository(head_dir, sub_dir):
    for root, dirs, files in os.walk(head_dir):
        for d in dirs:
            current_dir = os.path.join(root, d)
            if current_dir.endswith(sub_dir):
                return current_dir

# This function is used to find the newest version
def version_tuple(v):
    return tuple(map(int, (v.split("."))))

def get_newest_version(directory, current_version):
    newest_version = None
    available_versions = os.listdir(directory)
    for version in available_versions:
        if version_tuple(version ) > version_tuple(current_
version):
            newest_version = version

    return newest_version

# Finds the MyController object in the project
device = projects.primary.find("MyController", True)[0]

if not(device == None):
    device_identification = device.get_device_identification
()

    # Device Repository subdirectory
    sub_dir = os.path.join(str(device_identification.type),
device_identification.id)

    # Find the path where the device description is stored
    result_directory= find_device_repository(r"c:/", sub_
dir)

    if not result_directory == None:
        newest_version = get_newest_version(result_
directory, device_identification.version)

        if not newest_version == None:
            device.update(device_identification.type,
device_identification.id, newest_version)
```

System / User Interface (UI)

Browse Directory Dialog Box

Opens a dialog box for browsing a directory. In *--noUI* mode, you can enter a path here.

```
# Import the .NET class System.Environment from mscorlib.dll
first
# Otherwise the Environment.SpecialFolder Enumeration is not
available
```

```
from System import Environment
import os

# Set the necessary fields
dialog_message = "Select Python Example Directory"
preselected_path =  r"c:\Python"

# Store the selected path
selected_path = system.ui.browse_directory_dialog(dialog_
message, preselected_path, Environment.SpecialFolder.
Desktop, True)

# Remember to check if the user canceled the dialog
if(selected_path != None):
    # List files in the specified directory
    print(os.listdir(selected_path))
```

It consists of 4 parameters:

- a string containing the message
- a string containing the path that will be preselected when the dialog box opens
- the *Environment.SpecialFolder* contains the root folder for the browse dialog box
- a boolean parameter: If *True*, a button allowing you to create new folders is displayed in the dialog box.

This method returns the selected path. If you cancel the dialog box, nothing is returned.

Choose

This method allows you to choose between one of several listed items.

```
# Set the necessary fields
option_list = ["TM241C24R", "TM262M25MESS8T"]

dialog_message = "Select the controller which should be
added to the project:"

# A python tuple will be returned with the index of the
selected item or -1 if canceled
selected_item = system.ui.choose(dialog_message, option_
list, True)

# Remember to check if the user canceled the dialog
if(selected_item[0] != -1):
    print(option_list[selected_item[0]] + " will be added to
the project.")
```

It consists of 3 parameters:

- a string containing the message
- a list of options to be displayed: The objects are converted to string in order to display them.
- a boolean parameter: If *True*, a button allowing you to create new folders is displayed in the dialog box.

This method returns a Python tuple containing 2 items:

- the index of the selected item, or

  `-1` if cancelable was set to `True` and you canceled the dialog box
- the selected item or `None`

Error Detection

This method indicates an error detection message. It inhibits any further actions until the message has been acknowledged.

```
system.ui.error("Error")
```

Info

This method indicates an information message. It inhibits any further actions until the message has been acknowledged.

```
# Set the necessary fields
dialog_message = "Project update has been completed
successfully"

# This method is used to display a simple message to the
user
system.ui.info(dialog_message)
```

Open File Dialog Box

This method displays an **Open File** dialog box. In *--noUI* mode, you can enter a path here.

```
# Set the necessary fields
dialog_title = "Select a Project"
dialog_filename = None
initial_directory = None
file_filter = "(*.project)|*.project"

# Open a select file dialog which only accepts *.project
files
selected_file = system.ui.open_file_dialog(dialog_title,
dialog_filename, initial_directory, file_filter)

# Remember to check if the user canceled the dialog
if(selected_file != None):
    # Open the specified project
    if projects.primary:
        projects.primary.close()
    projects.open(selected_file)
```

Query String

This method queries the input or edit of a text string.

```
# Set the necessary fields
dialog_message = "Please enter a filename for your project:"
dialog_prefilled_text = "c:\Python\QueryStringExample.
project"

# This functions queries a string from the user
project_name = system.ui.query_string(dialog_message,
dialog_prefilled_text)

# If a project is opened, the project will be saved under
the new nameif(projects.primary):
    projects.primary.save_as(project_name)
```

It returns a string with the entered text.

Save File Dialog

This method displays a **Save File** dialog box. In *--noUI* mode, you can enter a path here.

```
import os

# Set the necessary fields
dialog_title = "Select a Filename"
initial_directory, dialog_filename = os.path.split(projects.
primary.path)
file_filter = "(*.project)|*.project"

# Open a select file dialog which only accepts *.project
files
selected_file = system.ui.save_file_dialog(dialog_title,
dialog_filename, initial_directory, file_filter)

# Remember to check if the user canceled the dialog
if(selected_file != None):
    # File was saved under the specified name, lets close
the project
    projects.primary.close()
```

Warning

This method indicates an advisory message. It inhibits any further actions until the message has been acknowledged.

```
# Set the necessary fields
dialog_message = "Project update failed. Please check the
log for detailed errors."

# This method is used to display a simple warning to the
user
system.ui.warning(dialog_message)
```

Reading Values

Overview

The script opens an application in EcoStruxure Machine Expert and logs in to the device. If the controller is not in RUNNING state, it will be set to RUNNING. Then the variable `iVar1` is read and displayed in the **Messages** view or command line. At the end, the application is closed.

Script Example *ReadVariable.py*

```
import os

# Define the necessary fields for this example
project_name = "Ohne Titel.project"
project_path = r"C:\Python"
value_name = "POU.testoutput"

try:
    # Clean up any open project
    if projects.primary:
        projects.primary.close()

    # Create the new project
    proj = projects.open(os.path.join(project_path, project_
name))

    # Set the active application
    app = proj.active_application

    # Create the online application
```

```
    onlineapp = online.create_online_application(app)

    # Login to device
    onlineapp.login(OnlineChangeOption.Try, True)

    # Set status of application to "run" if not in "run"
    if not onlineapp.application_state == ApplicationState.
run:
        onlineapp.start()

    # Wait 1 second
    system.delay(1000)

    # Read value of iVar1
    value = onlineapp.read_value(value_name)

    # Display value in message view or command line
    print (value_name + ": " + str(value))

    # Log out from device and close the project
    onlineapp.logout()
    #proj.close()
except Exception as exception:
    print("Error: " + str(exception))
    print("Please turn on the 'Script Tracing' function to
get detailed information about the script execution.")
```

Reading Values From Recipe and Send an Email

Overview

The script opens an application in EcoStruxure Machine Expert and logs in to the device. If the controller is not in mode RUNNING, it will be set to RUNNING. Then the variable `iVar1` is read and displayed in the **Messages** view or command line. At the end, the application is closed.

Script Example *ScriptEmail.py*

```
# Get the primary project reference
project = projects.primary
 # Retrieve active application
active_application = project.active_application

# Create online application
online_application = online.create_online_application
(active_application)

# Login to application
online_application.login(OnlineChangeOption.Try, True)

# Start PLC if necessary
if not online_application.application_state ==
ApplicationState.run:
    online_application.start()

# Wait 2 seconds
system.delay(2000)

# Define read values
watch_expressions = ["POU.value_1", "POU.value_2", "POU.
value_3"]

# Read values from the controller
```

```
watch_values = online_application.read_values(watch_
expressions)

# Open output file to write values
recipe_output_file = open(r"C:\Python\Objects\Output.txt",
"w")

for i in range(len(watch_expressions)):
    recipe_output_file.write(watch_expressions[i])
recipe_output_file.write(" = ")
recipe_output_file.write(watch_values[i])
recipe_output_file.write("\n")

# Close files
recipe_output_file.close()

# Send Email
# Import respective libraries
import smtplib
from email.mime.text import MIMEText
#open output file
recipe_output_file = open("C:\Python\Objects\Output.txt",
"r")
mail = MIMEText(recipe_output_file.read())
recipe_output_file.close()

# Email address sender and recipient
sender = "sender@sender.com"
to = "to@to.com"

# Set sender and recipient
mail["Subject"] = "Attention value has changed"
mail["From"] = sender
mail["To"] = to

# Send email
smtp = smtplib.SMTP("Name of the SMTP Server")
smtp.sendmail(sender, [to], mail.as_string())
smtp.quit()

# Logout and close application
online_application.logout()
project.close()
```

Determine Device Tree of the Open Project

Overview

This example determines the objects in the **Devices tree** of the open project and prints them out in the command line or **Messages** view. It can be started from the user interface or from the command line.

Script Example *DevicePrintTree.py*

```
# We enable the new python 3 print syntax
import sys

# Define the printing function
def printtree(treeobj, depth=0):
    if treeobj.is_root:
        name = treeobj.path
        deviceid = ""
    else:
```

```
            name = treeobj.get_name(False)
            if treeobj.is_device:
                deviceid = treeobj.get_device_identification()
            else:
                deviceid = ""
        print("{0} - {1} {2}".format(" "*depth, name, deviceid))
        for child in treeobj.get_children(False):
            printtree(child, depth+1)

# Now see whether a primary project is open.
if not projects.primary:
    print("Error: Please open a project file first!")
    sys.exit()

# And the actual output
print("Print the current tree of: ")
printtree(projects.primary)
```

Script Example 4: Import a Device in PLCOpenXML From Subversion

Overview

This example imports a device in PLCOpenXML from Subversion via command line `svn client`. It can be started from the user interface or from the command line.

Script Example *DeviceImportFromSvn.py*

```
# Imports a Device in PLCOpenXML from Subversion via command
line svn client
import os

# Some variable definitions
svn_executable = r"C:\Program Files\TortoiseSVN\bin\svn.exe"
project_file = r"C:\Python\test.project"

# !!! Important: File must be under version control !!!
export_file = r"C:\Python\SVN\MyController.xml"

# Clean up any open project:
if projects.primary:
    projects.primary.close()

# Fetch the PLCOpenXML data from subversion
# The with construct automatically closes the open pipe for
us.
with os.popen('"' + svn_executable + '" cat ' + export_file,
"r") as pipe:
    xmldata = pipe.read()

# Create a new project
project_reference = projects.create(project_file)

# Create the import reporter
class Reporter(ImportReporter):
    def error(self, message):
        system.write_message(Severity.Error, message)
    def warning(self, message):
        system.write_message(Severity.Warning, message)
    def resolve_conflict(self, obj):
        return ConflictResolve.Copy
    def added(self, obj):
```

```
        print("added: ", obj)
    def replaced(self, obj):
        print("replaced: ", obj)
    def skipped(self, obj):
        print("skipped: ", obj)
    @property
    def aborting(self):
        return False

# Create the importer instance
reporter = Reporter()

# Import the data into the project
project_reference.import_xml(reporter, xmldata)

# And finally save
project_reference.save()
```

Script Example 5: Creating and Editing POUs

Overview

The script *CreateDut.py* creates the objects *MyStruct*, *MyAlias*, and *MyUnion* in the project. The folder *DataTypes* must be available.

CreateDut.py

```
# encoding:utf-8
from __future__ import print_function
STRUCT_CONTENT = """\
    a : BOOL;
    b : BIT;
    c : BIT;
"""
UNION_WHOLE = """\
TYPE MyUnion :
UNION
    Zahl : INT;
    Prozent : MyAlias;
    Bits : MyStruct;
END_UNION
END_TYPE
"""
proj = projects.primary
folder = proj.find('DataTypes', recursive = True)[0]
# Create a struct DUT and insert the list of variables just
into the right
# place in line two, row 0 (line numbering starts with line
0)
struktur = folder.create_dut('MyStruct') # DutType.Structure
is the default
struktur.textual_declaration.insert(2, 0, STRUCT_CONTENT)
# Alias types get their "content" via the base type, which
will just end up
# as one line in the declaration part:
# TYPE MyAlias : INT (0..100); END_TYPE
bereich = folder.create_dut('MyAlias', DutType.Alias, "INT
(0..100)")
# Instead of injecting the variables into the existing
declaration,
# one can also just replace the complete declaration part,
including the
# boilerplate code.
```

```
union = folder.create_dut('MyUnion', DutType.Union)
union.textual_declaration.replace(UNION_WHOLE)
```

Script Example 6: User Interface / Interaction with the User

Overview

In some cases, scripts have to interact with the user. Simple APIs are provided for common interactions. The sample script *System_UI_Test.py* shows the possible functions.

System_UI_Test.py

```
# encoding:utf-8
from __future__ import print_function

"""Performs some tests on the messagestore and UI."""

print("Some Error, Warning and Information popups:")
system.ui.error("Fatal error: Everything is OK. :-)")
system.ui.warning("Your bank account is surprisingly low")
system.ui.info("Just for your information: 42")

print("Now, we ask the user something.")
res = system.ui.prompt("Do you like this?", PromptChoice.
YesNo, PromptResult.Yes);
print("The user selected '%s'" % res)

print("Now, the user can choose between custom options:")
res = system.ui.choose("Please choose:", ("First", 2, 7.5,
"Something else"))
print("The user selected option '%s'" % str(res)) # res is a
tuple

print("Now, the user can choose several options:")
res = system.ui.select_many("Please select one or more
options", PromptChoice.OKCancel, PromptResult.OK, ("La
Premiere", "The Second", "Das Dritte"))
print("The returned result is: '%s'" % str(res)) # res is a
tuple

print("Now, the user can select files and directories")
res = system.ui.open_file_dialog("Choose multiple files:",
filter="Text files (*.txt)|*.txt|Image Files(*.BMP;*.JPG;*.
GIF)|*.BMP;*.JPG;*.GIF|All files (*.*)|*.*", filter_index =
0, multiselect=True)
print("The user did choose: '%s'" % str(res)) # res is a
tuple as multiselect is true.

res = system.ui.save_file_dialog("Choose a file to save:",
filter="Text files (*.txt)|*.txt|Image Files(*.BMP;*.JPG;*.
GIF)|*.BMP;*.JPG;*.GIF|All files (*.*)|*.*", filter_index =
0)
print("The user did choose: '%s'" % res)

res = system.ui.browse_directory_dialog("Choose a
directory", path="C:\\")
print("The user did choose: '%s'" % res)

print("Now we query a single line string")
res = system.ui.query_string("What's your name?")
print("Nice to meet you, dear %s." % res)
```

```
print("Now we query a multi line string")
res = system.ui.query_string("Please tell me a nice story
about your life!", multi_line=True)
if (res):
    print("Huh, that has been a long text, at least %s
characters!" % len(res))
else:
    print("Hey, don't be lazy!")

print("Username and passwort prompts...")
res = system.ui.query_password("Please enter your favourite
password!", cancellable=True)
if res:
    print("Huh, it's very careless to tell me your favourite
password '%s'!" % res)
else:
    print("Ok, if you don't want...")

res = system.ui.query_credentials("Now, for real...")
if res:
    print("Username '%s' and password '%s'" % res) # res is
a 2-tuple
else:
    print("Sigh...")
```

Script Example 7: Manipulation of the Project Information Object

Overview

The script *ProjectInfoExample.py*, provides information in the **Project information** object. Important information items, such as **Title** and **Version**, have explicit properties. However, you can read and write the other information fields by using the dictionary syntax, for example, those for the properties of a library project.

The example below may seem unrealistic, but similar code is used in build servers that create, test, and possibly release automatic library projects and other projects. The ScriptEngine is one of the key elements for creating CI (Continuous Integration) and CD (Continuous Delivery) systems.

ProjectInfoExample.py

```
# encoding:utf-8
from __future__ import print_function

proj = projects.load("D:\Some.library")

info = proj.get_project_info()

# Set some values
info.company = "Test Library Ltd"
info.title = "Script Test Project"
info.version = (0, 8, 15, 4711)
info.default_namespace = "testlibrary"
info.author = "Python von Scriptinger"

# some values recommended in the library toolchain
info.values["DefaultNamespace"] = "testlibrary"
info.values["Placeholder"] = "testlibrary"
info.values["DocFormat"] = "reStructuredText"

# now we set a custom / vendor specific value.
info.values["SpecialDeviceId"] = "PLC0815_4711"
```

```
# Enable generation of Accessor functions, so the IEC
# application can display the version in an info screen.
info.change_accessor_generation(True)

# And set the library to released
info.released = True;

proj.save()
```

Script Example 8: Trace Scripting

Overview

The following examples are intended as a supplement to the automatically generated API documentation from CODESYS Scripting to CODESYS Trace.

Creating a Trace

Use the Trace Scripting API to create a new trace.

```
myApplication = projects.primary.find("Device", "PLC Logic", "Application")[0]
traceObject = trace.create(myApplication, "MyTraceName")
```

As a prerequisite, a task named `Task` must already exist. If it does not exist, a *ValueError* is detected.

```
traceObject.task_name = "Task"
```

In the script, specify in which task the newly created trace is to be executed.

```
myApplication = projects.primary.find("Device", "PLC Logic", "Application")[0]
traceObjectWithTaskSet = trace.create(myApplication, "NameOfMySecondTrace",
"MainTask")
assert traceObjectWithTaskSet.task_name == "MainTask"
```

Finding and Verifying an Existing Trace

To find an existing trace, use the `project.find` method.

For the other objects of the `IScriptObject` type, `.is_trace_object` equals `FALSE`.

```
foundObjects = projects.primary.find("Device", "PLC Logic", "Application",
"MyTraceObject")
expectedTraceObject = foundObjects[0]
assert expectedTraceObject.is_trace_object, "Could not find trace object with the
expected name"
```

Adding and Modifying Trace Variables

Use the `add_variable()` method to add a new `ScriptTraceVariable` and set the properties during creation.

```
myTrace = projects.primary.find("Device", "PLC Logic", "Application",
"MyTraceObject")[0]

addedVariable = myTrace.add_trace_variable(variableName="PLC_PRG.a1",
        graphColor= 0xff000000,
        graphType=GraphType.LINES,
        activateMaxWarning=True,
        maxWarningArea=9.0,
        maxColor= 0xffff0000)
addedVariable.enabled = False

for i in range(2,11):
        newVariable = myTrace.variable_list.add()
```

```
        newVariable.variable_name = "PLC_PRG.a"+str(i)
        newVariable.graph_type = GraphType.LINES_CROSSES

assert len(myTrace.variable_list) == 10
```

Modifying the `ScriptTrace` Variable List

Removing a specific `ScriptTrace` variable.

In the following example, a variable with the specified name is found and removed from the `ScriptTraceVariableList`.

```
def remove_variable_by_name(traceObject, variableName):
        index_to_remove = -1
        for variable in traceObject.variable_list:
                if variable.variable_name == variableName:
                        index_to_remove = traceObject.variable_list.index_of
(variable)
                        break
        if index_to_remove != -1:
        traceObject.variable_list.remove(index_to_remove)
```

Modifying the Recording Settings

The following examples are based on the example provided in Creating a Trace, page 870.

- Setting the resolution (ms/µs) of the trace recording:
  `traceObject.resolution = Resolution.MicroSeconds`

  Use `Resolution.MilliSeconds` to perform the recording based on milliseconds.
- Setting the recording condition:
  `traceObject.record_condition = "PLC_PRG.bDoRecord"`

  For further information, refer to Trace Configuration, page 460.
- Setting a comment:
  `traceObject.comment = "This trace records the ..."`
- Setting the `auto_start` option:
  `traceObject.auto_start = True`
- Setting the option to record in every n^{th} cycle:
  `traceObject.every_n_cycles = 10`

  For further information, refer to Advanced Trace Settings, page 468.

`ScriptTrace` Diagrams and `ScriptTrace` Variables

Two options are available for managing `ScriptTrace` diagrams and `ScriptTrace` variables.

Option 1: Creating one `ScriptTrace` diagram with all `ScriptTrace` variables.

The following example code creates a simple diagram where all variables are used.

First step: A `ScriptTrace` object and some `ScriptTrace` variables are created:

```
traceObject = projects.primary.find("Device","PLC Logic","Application","Trace")[0]
assert traceObject.is_trace_object
for i in range(5):
        traceObject.add_trace_variable(variableName="PLC_PRG.a"+str(i))
```

Second step: A `ScriptTrace` diagram is created and all `ScriptTrace` variables are added afterwards:

```
traceDiagram = traceObject.diagrams.add()
for traceVar in traceObject.variable_list:
```

```
    traceDiagram.add_diagram_variable(traceVar)
```

Option 2: Creating one `ScriptTrace` diagram per `ScriptTrace` variable.

The following Python method takes one `ScriptTrace` object and adds a diagram per `ScriptTrace` variable:

```
def one_diagram_per_variable(traceObject):
    assert traceObject.is_trace_object
    for traceVar in traceObject.variable_list:
        diagram = traceObject.diagrams.add(traceVar)
        diagram.name = traceVar.variable_name
```

Copying the Settings of a `ScriptTrace` Diagram

The following example indicates how to modify and copy the settings of a `ScriptTrace` diagram.

```
traceObject = projects.primary.find("Device","PLC Logic","Application","Trace")[0]
assert traceObject.is_trace_object

# apply settings for the first diagram
traceObject.diagrams[0].y_axis.mode = AxisScaleMode.FixedLength
traceObject.diagrams[0].y_axis.range = 10.0
traceObject.diagrams[0].y_axis.color = 0xffee0000
traceObject.diagrams[0].y_axis.draw_grid = True
traceObject.diagrams[0].y_axis.grid_color = 0xffee0000
traceObject.diagrams[0].tickmark_fixed_spacing = True
traceObject.diagrams[0].tickmark_fixed_distance = 5.0
traceObject.diagrams[0].tickmark_fixed_subdivisions = 4

# copy the settings from the first diagrams to all other diagrams
for i in range(1,len(traceObject.diagrams))
    traceObject.diagrams[i].y_axis.copy_from(traceObject.diagrams[0])
```

Online Handling of Trace

Downloading and starting a trace.

The following example finds the `Trace` trace of the `Application` application, downloads it, and starts it.

```
proj = projects.open(r"<Path to the project file>")
app = proj.find("Device", "PLC Logic", "Application")[0]
with online.create_online_application(app) as onlineapp:
    traceObject = proj.find("Device", "PLC Logic", "Application", "Trace")[0]
    onlineapp.login(OnlineChangeOption.Never, True)
    onlineapp.start()
    editor = traceObject.open_editor()
    editor.download()
    editor.start()
```

Saving the trace to a file:

The following example is based on the *Downloading and starting a trace* example.

A trace must be stopped before it can be saved. This can be done either by reaching the trigger condition or by an explicit call of `stop()`.

It is assumed that a trace is already running.

```
editor.stop()
editor.save(r"<File path to csv file>")
```

Querying various information about the trace recording:

The following example is based on the *Downloading and starting a trace* example.

It is assumed that a trace is already running.

```
print("Packet State: {}".format(editor.get_packet_state()))
print("Trace started at {} (absolute timestamp)".format(editor.get_trace_start_
timetamp()))
```

Trigger Handling

Configuring a Boolean trigger:

```
traceObject = proj.find("Device", "PLC Logic", "Application", "Trace")[0]
traceObject.trigger_variable = "PLC_PRG.bVar"
traceObject.trigger_edge = TriggerEdge.Positive
traceObject.post_trigger_samples = 20
```

Although the trigger level is only required for numeric trigger variables, it is a good practice to explicitly set the trigger to `None`.

```
traceObject.trigger_level = None
```

Configuring a numeric trigger:

```
traceObject = proj.find("Device", "PLC Logic", "Application", "Trace")[0]
traceObject.trigger_variable = "PLC_PRG.iTemperature"
traceObject.trigger_edge = TriggerEdge.Positive
traceObject.post_trigger_samples = 20
```

If the trigger variable is of type REAL or LREAL, then the trigger level can also be specified as a floating point number (for example, `80.5`).

```
traceObject.trigger_level = None
```

Waiting for the trigger and saving recorded data:

The following example is based on the *Downloading and starting a trace* example and on the *Configuring a * trigger* examples.

```
triggerstate = editor.get_trigger_state()
while triggerstate != TriggerState.TriggerReached:
        system.delay(200)
        triggerstate = editor.get_trigger_state()

editor.save(r"<File path to csv file>"
```

Querying timestamp information after reaching the trigger:

The following example is based on the *Downloading and starting a trace* example and on the *Configuring a * trigger* examples.

```
print("Trigger reached at {} (absolute timestamp)".format(editor.get_trigger_
timetamp()))
print("Trigger reached at {}".format(editor.get_trigger_startdate()))
print("Trace was running {}ms until trigger has been reached".format(editor.get_
trigger_timetamp() - editor.get_trace_start_timetamp()))
```

Resuming trace recording after reaching the trigger:

The following example is based on the *Downloading and starting a trace* example and on the *Configuring a * trigger* examples.

```
editor.reset_trigger()
```

DeviceTrace

Creating a `DeviceTrace`:

```
myDevice = projects.primary.find("Device")[0]
devicetrace = trace.create(myDevice, "DeviceTrace")
```

Retrieving trace recordings from the device:

The following example is based on the *Creating a* `DeviceTrace` example.

```
traceObject = projects.primary.find("Device", "DeviceTrace")[0]
```

For application traces, the result of `get_online_traces` contains only traces of the application.

```
# traceObject = projects.primary.find("Device", "PLC Logic", "Application",
"Trace")[0]
editor = traceObject.open_editor()
for tracepacket in editor.get_online_traces():
    print(tracepacket)
```

Uploading an existing trace as `DeviceTrace`:

The following example is based on the *Creating a* `DeviceTrace` example.

```
traceObject = projects.primary.find("Device", "DeviceTrace")[0]
editor = traceObject.open_editor()
editor.upload_to_device_trace("CpuCoreLoad")
```

Advanced Example: Checkout a Library from SVN and Installation

Overview

The following sample script can perform the checkout and installation of a library as part of a CT (Continuous Testing) environment so that they can be tested. The SVN Add-on must be installed on the PC and the SVN Add-on license must be activated.

Sample Script

```
import tempfile

if projects.primary:
    projects.primary.close()

tempdir = tempfile.mkdtemp()
URL = "svn://localhost/testrepo/trunk/SvnTestLibrary/"

proj = svn.checkout(URL, tempdir, "testlibrary", as_library=
True)
proj.save()

repo = librarymanager.repositories[0]
librarymanager.install_library(proj.path, repo, True)

proj.close()
```

Managing Device User Rights Using the Scripting API

Managing Device User Rights Using the Scripting API

Scripting for Using Online Services

EcoStruxure Machine Expert provides access to many of their online services via the scripting API. In order to establish a connection or to use an online service at a later time, valid credentials must be stored in the system.

Providing Specific Credentials for Online Services

You can store credentials via online device or online application in case of multi-controller projects. If there are specific credentials provided for the connection, they will be used by the system.

Example:

```
# create an "online device" to use online services
root_device = projects.primary.find("LMC_PacDrive", False)
[0]
online_device = online.create_online_device(root_device)

# store credentials specific to this "online device"
online.set_specific_credentials(online_device, "my_user",
"my_password")

# use of any online service
online_device.connect()
```

Providing Default Credentials for Online Services

If no specific credentials are provided for the connection, the system uses the default credentials.

Example:

```
# create an "online device" to use online services
root_device = projects.primary.find("LMC_PacDrive", False)
[0]
online_device = online.create_online_device(root_device)

# store default credentials
online.set_default_credentials("my_user", "my_password")

# use of any online service
online_device.connect()
```

Scripting for Enforced Password Renewal

The following scenarios require the password to be changed by the user after authentication:

- First login to a new controller.
- First connection after the user rights management has been reset to default.
- A password renewal is enforced for a specific user by an administrator of the device.
- A password expired conforming to the expiration timeout configured by your controller.

EcoStruxure Machine Expert does not support the renewal of passwords using the scripting API. Perform this by using the service tool Controller Assistant.

You can call the latest version of Controller Assistant from command line as indicated in the following example:

```
"c:\Program Files (x86)\Schneider Electric
\EcoStruxureMachine Expert\Tools\ControllerAssistant
\ControllerAssistant.exe" -username Administrator -password
Administrator -renewalpassword MyNewPassword
-getcontrollerinfo etcp4://192.168.3.50
```

For further information, refer to the Controller Assistant User Guide.

Controller Feature Sets for Migration

What's in This Chapter

Controller Feature Sets for Migration 876

Controller Feature Sets for Migration

Twido Controllers

Controller	Dig In	Dig Out	MOD	FC	HSC	PWM	Serial	ETH
TWDLCAA10DRF	6	4	No	3	1	0	1	No
TWDLCDA10DRF	6	4	No	3	1	0	1	No
TWDLCAA16DRF	9	7	No	3	1	0	1+1	No
TWDLCDA16DRF	9	7	No	3	1	0	1+1	No
TWDLCAA24DRF	14	10	No	3	1	0	1+1	No
TWDLCDA24DRF	14	10	No	3	1	0	1+1	No
TWDLCAA40DRF	24	16	No	4	2	2	1+1	No
TWDLCDA40DRF	24	16	No	4	2	2	1+1	No
TWDLCAE40DRF	24	16	No	4	2	2	1+1	Yes
TWDLCDE40DRF	24	16	No	4	2	2	1+1	Yes
TWDLMDA20DTK	12	8	Yes	2	2	2	1+1	No
TWDLMDA20DUK	12	8	Yes	2	2	2	1+1	No
TWDLMDA20DRT	12	8	Yes	2	2	2	1+1	No
TWDLMDA40DTK	24	16	Yes	2	2	2	1+1	No
TWDLMDA40DUK	24	16	Yes	2	2	2	1+1	No

Dig In = number of digital inputs

Dig Out = number of digital outputs

MOD = expansion modules

FC = number of fast counters

HSC = number of high-speed counters

PWM = number of pulse generators

Serial = number of serial ports

ETH = Ethernet ports

M221 Controllers

Controller	Dig In	Dig Out	Ana In	MOD	FC	HSC	PWM	Serial	ETH	CART
TM221C16R	9	7	2	TM2/ TM3	4	2	0	1	No	1
TM221C16T	9	7	2	TM2/ TM3	4	2	2	1	No	1
TM221C24R	14	10	2	TM2/ TM3	4	2	0	1	No	1
TM221C24T	14	10	2	TM2/ TM3	4	2	2	1	No	1
TM221C40R	24	16	2	TM2/ TM3	4	2	0	1	No	2

Controller	Dig In	Dig Out	Ana In	MOD	FC	HSC	PWM	Serial	ETH	CART
TM221C40T	24	16	2	TM2/ TM3	4	2	2	1	No	2
TM221CE16R	9	7	2	TM2/ TM3	4	2	0	1	Yes	1
TM221CE16T	9	7	2	TM2/ TM3	4	2	2	1	Yes	1
TM221CE24R	14	10	2	TM2/ TM3	4	2	0	1	Yes	1
TM221CE24T	14	10	2	TM2/ TM3	4	2	2	1	Yes	1
TM221CE40R	24	16	2	TM2/ TM3	4	2	0	1	Yes	2
TM221CE40T	24	16	2	TM2/ TM3	4	2	2	1	Yes	2
TM221M16R/G	8	8	2	TM2/ TM3	4	2	0	2	No	0
TM221M16T/G	8	8	2	TM2/ TM3	4	2	2	2	No	0
TM221M32TK	16	16	2	TM2/ TM3	4	2	2	2	No	0
TM221ME16R/G	8	8	2	TM2/ TM3	4	2	0	1	Yes	0
TM221ME16T/G	8	8	2	TM2/ TM3	4	2	2	1	Yes	0
TM221ME32TK	16	16	2	TM2/ TM3	4	2	2	1	Yes	0

Dig In = number of digital inputs

Dig Out = number of digital outputs

Ana In = number of analog inputs

MOD = expansion modules

FC = number of fast counters

HSC = number of high-speed counters

PWM = number of pulse generators

Serial = number of serial ports

ETH = Ethernet ports

CART = number of cartridges

EcoStruxure Machine Expert Controllers

Controller	Dig In	Dig Out	Ana In	MOD	FC	HSC	PWM	Serial	ETH	CART
TM241C24R	14	10	0	TM2/ TM3	8	2	4	2	No	1
TM241C24T/U	14	10	0	TM2/ TM3	8	2	4	2	No	1
TM241C40R	24	16	0	TM2/ TM3	8	2	4	2	No	2
TM241C40T/U	24	16	0	TM2/ TM3	8	2	4	2	No	2
TM241CE24R	14	10	0	TM2/ TM3	8	2	4	2	Yes	1
TM241CE24T/U	14	10	0	TM2/ TM3	8	2	4	2	Yes	1
TM241CE40R	24	16	0	TM2/ TM3	8	2	4	2	Yes	2

Controller	Dig In	Dig Out	Ana In	MOD	FC	HSC	PWM	Serial	ETH	CART
TM241CE40T/U	24	16	0	TM2/ TM3	8	2	4	2	Yes	2
TM241CEC24R	14	10	0	TM2/ TM3	8	2	4	2	Yes	1
TM241CEC24T/U	14	10	0	TM2/ TM3	8	2	4	2	Yes	1
HMISCU•A5	16	10	0	No	2	1	2	1	Yes	0
HMISCU•B5	8	8	2	No	2	1	2	1	Yes	0

Dig In = number of digital inputs

Dig Out = number of digital outputs

Ana In = number of analog inputs

MOD = expansion modules

FC = number of fast counters

HSC = number of high-speed counters

PWM = number of pulse generators

Serial = number of serial ports

ETH = Ethernet ports

CART = number of cartridges

Glossary

B

Branch:

A branch is a copy of a project where you can work on changes separately without affecting the main version.

C

CFC:

(*continuous function chart*) A graphical programming language (an extension of the IEC 61131-3 standard) based on the function block diagram language that works like a flowchart. However, no networks are used and free positioning of graphic elements is possible, which allows feedback loops. For each block, the inputs are on the left and the outputs on the right. You can link the block outputs to the inputs of other blocks to create complex expressions.

Commit:

A commit is the action of saving changes to a project with a message that describes what was modified.

D

DTM:

(*device type manager*) Classified into 2 categories:

- Device DTMs connect to the field device configuration components.
- CommDTMs connect to the software communication components.

The DTM provides a unified structure for accessing device parameters and configuring, operating, and diagnosing the devices. DTMs can range from a simple graphical user interface for setting device parameters to a highly sophisticated application capable of performing complex real-time calculations for diagnosis and maintenance purposes.

DUT:

(*data unit type*) Along with the standard data types the user can define own data type structures, enumerationen types, and references as data type units in a DUT editor.

E

element:

The short name of the ARRAY element.

F

FBD:

(*function block diagram*) One of 5 languages for logic or control supported by the standard IEC 61131-3 for control systems. Function block diagram is a graphically oriented programming language. It works with a list of networks, where each network contains a graphical structure of boxes and connection lines, which represents either a logical or arithmetic expression, the call of a function block, a jump, or a return instruction.

FBS:

(*file-based storage*) A file format for saving EcoStruxure Machine Expert projects. The project is stored as a folder structure and each object of the project is stored as an individual file in a textual (editable) format.

The folder structure can consist of several folder levels with long folder names which can lead to long path names. To allow long path names, perform the following actions on the computer running EcoStruxure Machine Expert:

- In the **Windows Settings**, activate the option **Enable long paths**. Verify whether this is allowed by your company rules or adapt the Group Policy accordingly.
- Adapt the Git settings with the following command line argument:
  `git config --global core.longpaths true`

G

GitHub:

GitHub is an online portal that allows individuals and teams to collaborate on software projects by sharing and managing code in centralized repositories.

GRAFCET:

The functioning of a sequential operation in a structured and graphic form.

This is an analytical method that divides any sequential control system into a series of steps, with which actions, transitions, and conditions are associated.

GVL:

(*global variable list*) Manages global variables within an EcoStruxure Machine Expert project.

I

IL:

(*instruction list*) A program written in the language that is composed of a series of text-based instructions executed sequentially by the controller. Each instruction includes a line number, an instruction code, and an operand (refer to IEC 61131-3).

L

LD:

(*ladder diagram*) A graphical representation of the instructions of a controller program with symbols for contacts, coils, and blocks in a series of rungs executed sequentially by a controller (refer to IEC 61131-3).

M

MAC address:

(*media access control address*) A unique 48-bit number associated with a specific piece of hardware. The MAC address is programmed into each network card or device when it is manufactured.

O

OPC UA:

OPC Unified Architecture: OPC UA is an interoperability standard for the secured and reliable exchange of data in the industrial automation space. It is a platform independent communication protocol using the server/client model. The connection between client and server is commonly based on the reliable transport layer protocol (TCP, Transmission Control Protocol).

For more information about the OPC especially OPC UA refer to the official webpage of the OPC Foundation at https://opcfoundation.org.

P

POU:

(*program organization unit*) A variable declaration in source code and a corresponding instruction set. POUs facilitate the modular re-use of software programs, functions, and function blocks. Once declared, POUs are available to one another.

Python interpreter:

A program that directly executes instructions that have been entered.

R

REPL:

(*read-eval-print-loop*) A simple, interactive computer programming environment. It takes single user inputs, evaluates them, and returns the result.

Repository:

A repository is a storage location that acts like a project folder or workspace, where code and its version history are organized and managed.

RTS:

(*request to send*) A data transmission signal and CTS signal that acknowledges the RTS from the destination node.

S

Sercos:

(*serial real-time communications system*) A digital control bus that interconnects, motion controls, drives, I/Os, sensors, and actuators for numerically controlled machines and systems. It is a standardized and open controller-to-intelligent digital device interface, designed for high-speed serial communication of standardized closed-loop real-time data.

SFC:

(*sequential function chart*) A language that is composed of steps with associated actions, transitions with associated logic condition, and directed links between steps and transitions. (The SFC standard is defined in IEC 848. It is IEC 61131-3 compliant.)

shell:

A shell is a (console) program that allows the user to interact with a program with a textual based interface.

Solution:

Container that groups related Machine Expert projects and files into a single workspace for organizing and managing configurations.

T

TLS:

(*transport layer security*) A protocol for encrypted data transfer. Successor of *SSL*.

U

UDP:

(*user datagram protocol*) A connectionless mode protocol (defined by IETF RFC 768) in which messages are delivered in a datagram (data telegram) to a destination computer on an IP network. The UDP protocol is typically bundled with the Internet protocol. UDP/IP messages do not expect a response, and are therefore ideal for applications in which dropped packets do not require retransmission (such as streaming video and networks that demand real-time performance).

UTC:

(*universal time coordinated*) The primary time standard by which the world regulates clocks and time.

Index

A

ABS
IEC operator ... 719
access protection ... 158
Acknowledge diagnosis command ... 37
Acknowledge Diagnosis for Subtrees command ... 37
ACOS
IEC operator ... 724
ADD
IEC operator ... 679
Add latch var column ... 490
Add message column ... 490
adding controllers ... 60
adding controllers by drag and drop ... 58
adding devices and modules by drag and drop ... 59
adding devices from device template by drag and drop ... 59
adding expansion devices by drag and drop ... 58
addressing ... 804
ADR
operator ... 703
alarm class editor ... 490
Alarm Configuration object ... 487
alarm group editor ... 495
alias data type ... 668
analog inputs
CANopen ... 791
Analyzation library ... 352
analyzing expressions ... 352
AND
IEC operator ... 689
ANY ... 640
ANY_<type> ... 640
ANY_NUM_TO
IEC operator ... 718
ANY_TO
IEC operator ... 718
Applications Tree ... 41
array ... 674
array declaration ... 375
ASCII ... 642
ASIN
IEC operator ... 723
ATAN
IEC operator ... 724
atomic operations ... 421
Attribute monitoring_display ... 618
Attribute monitoring_encoding ... 618
automatic I/O mapping ... 147
axis allocation
cam motion editor ... 532, 534

B

backend services ... 34
backtick identifiers ... 552
BIT ... 645
BITADR
operator ... 704
BOOL ... 635
BOOL_TO
IEC operator ... 708
boot application ... 239
Boot Application ... 235
build-time performance ... 794
buttons for navigation within editors ... 32

C

CAL
IEC operator ... 705
calculate value automatically
cam motion editor ... 532
cam data
cam data structure ... 530
cam data structures
generate data ... 531
generation ... 529
cam diagram
add ... 529
cam motion editor
generating cam data structures ... 529
parameterizing cam diagrams ... 529
cam segment ... 529
CANopen analog inputs ... 791
CANopen devices ... 62
CASE
instruction ... 366
Catalog view ... 45
certificates
expired ... 249
CFC execution order ... 322
character constant UCHAR ... 753
character sets ... 642
code page ... 642
command
Convert Device ... 67
Convert SoMachine Basic Project ... 69
Convert Twido Project ... 69
communication manager configuration ... 62
communication settings ... 95
Communication Settings in **Controller selection mode** ... 87
__COMPARE_AND_SWAP
operator ... 743
comparing programming objects ... 42
conditionalshow ... 601
conditionalshowsymbols ... 602
configuration diagnostic ... 65
constant UCHAR ... 753
constants ... 574
Content
operator ... 704
CONTINUE
instruction ... 369
controller - HMI variable exchange ... 525
conversions
overloaded ... 707
Convert Device command ... 67
convert SoMachine Basic project ... 69
convert Twido project ... 69
core dump ... 256
COS
IEC operator ... 722
CPU Load tab ... 422
__CURRENTTASK
operator ... 742
cursor positions within editors ... 32

D

data type
VERSION ... 670
data types ... 635
DATE ... 639
date and time data types ... 639
DATE_AND_TIME ... 639
DATE_TO
IEC operator ... 714
declaring
arrays ... 375
__DELETE
operator ... 726
device templates ... 770
device user management
access protection ... 158
device user rights
disabling ... 167
resetting ... 167
device user rights management ... 167
devices
adding ... 63
Devices tree ... 37
diagnostic messages for devices ... 37
dialog box
Select Function Block ... 155
`dir(...)`
Python function ... 822
disabling encryption ... 171
discontinuous courses of the positions ... 533
distributing tasks over multiple processor cores ... 417
DIV
IEC operator ... 683
DOS Latin 1 ... 642
double Nodenames ... 801
download ... 234, 239
boot application ... 239
Boot Application ... 235
DT ... 639
DT_TO
IEC operator ... 714
DTM ... 37

E

EcoStruxure Machine Expert Twin ... 144
Edit communication settings dialog box ... 95
editor navigation ... 32
Emulation Settings ... 144
encryption
disabling ... 171
END_IF
instruction ... 365
`END_VAR` ... 566
END_WHILE
instruction ... 368
`EQ`
IEC operator ... 702
executing Python scripts ... 546
execution order
CFC ... 322
EXIT
instruction ... 369
`EXP`
IEC operator ... 721
expansion devices ... 61
expansions ... 61
expired certificates ... 249
Export Alarms ... 490
exporting alarms ... 498
`EXPT`
IEC operator ... 725
external variables - `VAR_EXTERNAL` ... 570

F

FAQ
login to controller not successful ... 800
FB_Exit ... 579
FB_init
IEC operator ... 741
FB_Init ... 579
FB_MultiCam ... 530
FB_Reinit ... 579
FdtConnections node ... 37
FFB Finder ... 426
field device configuration ... 62
fieldbus health information ... 64
fieldbusses supported by templates ... 764
file-based storage ... 42, 46, 519, 833
firewall settings ... 158
first login to controller ... 162
FOR
instruction ... 367
Function and Function Block Finder ... 426
function templates ... 779
Functional Model node ... 42
Functional tree ... 42

G

`GE`
IEC operator ... 701
Generated IEC source code tab of the cam motion editor ... 532
global node operator ... 741
global variables - `VAR_GLOBAL` ... 569
`GT`
IEC operator ... 699

H

health information
fieldbus ... 64
history of cursor positions within editors ... 32
HMI - controller variable exchange ... 525
HMI connection indications ... 527

I

IEC code
generating from cam data ... 530
IEC objects
fieldbus Diagnostic/O mapping ... 64
IEC Objects view ... 130
IEC source code
generating ... 544
IEC source code of the cam diagram ... 532
IF
instruction ... 365
implicit enumeration ... 665
`IMPLICIT_ENUM` ... 665
Import Alarms ... 490
importing alarms ... 498
increasing build-time performance ... 794

indications
HMI connection 527
INI
IEC operator 741
input and output variables - VAR_IN_OUT 567
input data structure
FB_MultiCam 530
MC_Cam_ID 530
input variables - VAR_INPUT 567
inspectapi.dir
Python function 822
installation
third-party Sercos devices 56
instance variables - VAR_INST 570
instructions
ST editor 362
INT 636
INT_TO
IEC operator 711
integer 636
IronPython 809
__ISVALIDREF
operator 727

J

JMP
instruction 370

L

lambda
law of motion 538
Latin 1 642
Latin 1 encoding 642
law of motion 538
LE
IEC operator 700
LIMIT
IEC operator 698
literals
typed 577
LN
IEC operator 720
Local Network option 95
local variables - END_VAR 566
local variables - VAR 566
LOG
IEC operator 720
login 234, 239
login to controller 162
login to controller not successful 800
Logoff current device user command 163
logoff procedure 163
LREAL 637
LREAL_TO
IEC operator 711
LT
IEC operator 700
LTIME 639

M

managing tags 45
MAX
IEC operator 697
MC_Cam_ID 530
memory usage by recipes 453
menus 793
MIN
IEC operator 697
MOD
IEC operator 685
Modbus IOScanner on a Serial Line 794
Modbus Serial IOScanner
error detected 794
Modbus SL devices 62
Modbus slave disconnected 794
Modules view 35
monitoring_display 618
monitoring_encoding 618
motion design object editor 383
Motion Design Object... command 383
Motion Editor
displaying several **Motion Editor** views 530
Motion Editor tab 534
MOVE
IEC operator 686
MUL
IEC operator 681
multicore 417
CPU Load tab 422
global variable list (tasklocal) 207
multicore tools 420
multiple processor cores 417
MUX
IEC operator 698

N

Navigate backward button 32
Navigate forward button 32
Navigate to button 32
navigation within editors 32
NE
IEC operator 702
Network Device Identification
accessing new controllers 796
connecting via IP address and address information 798
FAQs 799
network variables list (receiver) 393
__NEW
operator 728
NodeName 98
Nodenames
double 801
NOT
IEC operator 691
NVL
configuration example 397
considerations 389
controllers supporting NVL 389
network variables list 387
rules 394
NVL communication suspended 795

O

online mode
cam motion editor 532
OR
IEC operator 689
output variables - VAR_OUTPUT 567
overloaded conversions 707

P

parameterizing cam diagrams 529
persistent variables 573
__POOL operator 738
"__POOL." operator 741
pragmas
  region 597
PROGRAM 177
programming environments for Python 809
project archives 264
protecting applications 165
publishing variables 521
publishing variables (HMI) 523
Python 809
Python script execution 546

Q

__QUERYINTERFACE
  operator 730
__QUERYPOINTER
  operator 731

R

REAL 637
REAL_TO
  IEC operator 711
recipes
  memory usage 453
refreshing variables 525
region pragmas 597
remanent variables 571
Remove latch var column 490
Remove message column 490
REPEAT
  instruction 369
retain variables 572
RETURN
  instruction 364
ROL
  IEC operator 693
ROR
  IEC operator 694
routing 804
run 250

S

Script Engine 809
Script Hook Manager 546
scripting for managing user rights 874
searching within catalogs 45
SEL
  IEC operator 695
Select Function Block dialog box 155
selecting variables 522
Sercos
  installation of third-party devices 56
Service Status message 34
services 34
Services icon 34
SHL
  IEC operator 691
shortcuts 793
SHR
  IEC operator 692
SIN
  IEC operator 721
SIZEOF
  IEC operator 687
smart coding 634
SmartTag functions 677
SQRT
  IEC operator 719
ST editor
  instructions 362
startup performance 792
static variables - VAR_STAT 569
Status & Staging view 46
stop 250
STRING 638
STRING_TO
  IEC operator 715
SUB
  IEC operator 682
symbol configuration 512

T

tagging catalog items 45
TAN
  IEC operator 723
tasklocal global variable list 207
tasks
  adding 233
  distribution over multiple processor cores 417
template libraries 770
templates 763
temporary variables - VAR_TEMP 569
TEST_AND_SET
  operator 745
TIME 639
time data types 639
TIME_OF_DAY 639
  IEC operator 712
TIME_TO
  IEC operator 712
TO_BOOL
  IEC operator 709
TOD 639
Tools Tree 42
transfer variable - VAR_IN_OUT CONSTANT 568
transition 190
Troubleshooting 791
TRUNC
  IEC operator 716
TRUNC_INT
  IEC operator 717
typed literals 577

U

UCHAR constant 753
Unicode 642
UNION 644
updating devices 66
Use DTM Connection checkbox 37
UTF-16 643
UTF-16 encoding 643
UTF-8 encoding 642
UTF-8 monitoring 618
UTF8# string 753
__UXINT 636
__UXINT data type 688

V

VAR 566
VAR_EXTERNAL 570
VAR_GENERIC CONSTANT 575
VAR_GLOBAL 569
VAR_IN_OUT 567
VAR_IN_OUT CONSTANT 568
VAR_INPUT 567
VAR_INST 570
VAR_OUTPUT 567
VAR_STAT 569
VAR_TEMP 569
variable definition 517
variable exchange
  communication speed 525
variable types 517
variables 566
  persistent 218
  publishing 521
  publishing Vijeo-Designer 523
  remanent 218
VERSION data type 670
Vijeo-Designer
  publishing variables 523
visualizations 773

W

WHILE
  instruction 368
Windows 1252 642
WSTRING 639
WSTRING encoding 643

X

__XADD
  operator 744
__XINT 636
XOR
  IEC operator 690
XSIZEOF
  IEC operator 688
__XWORD 636

Schneider Electric
1884 Boulevard de la Défense
92000 Nanterre
France

se.com

© 2026 Schneider Electric. All rights reserved.

EIO0000002854.10