

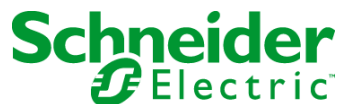
SoMachine

Programming Guide

02/2018

EIO00000000067.15

www.schneider-electric.com



The information provided in this documentation contains general descriptions and/or technical characteristics of the performance of the products contained herein. This documentation is not intended as a substitute for and is not to be used for determining suitability or reliability of these products for specific user applications. It is the duty of any such user or integrator to perform the appropriate and complete risk analysis, evaluation and testing of the products with respect to the relevant specific application or use thereof. Neither Schneider Electric nor any of its affiliates or subsidiaries shall be responsible or liable for misuse of the information contained herein. If you have any suggestions for improvements or amendments or have found errors in this publication, please notify us.

You agree not to reproduce, other than for your own personal, noncommercial use, all or part of this document on any medium whatsoever without permission of Schneider Electric, given in writing. You also agree not to establish any hypertext links to this document or its content. Schneider Electric does not grant any right or license for the personal and noncommercial use of the document or its content, except for a non-exclusive license to consult it on an "as is" basis, at your own risk. All other rights are reserved.

All pertinent state, regional, and local safety regulations must be observed when installing and using this product. For reasons of safety and to help ensure compliance with documented system data, only the manufacturer should perform repairs to components.

When devices are used for applications with technical safety requirements, the relevant instructions must be followed.

Failure to use Schneider Electric software or approved software with our hardware products may result in injury, harm, or improper operating results.

Failure to observe this information can result in injury or equipment damage.

© 2018 Schneider Electric. All Rights Reserved.

Table of Contents



	Safety Information	17
	About the Book	19
Part I	Introduction	25
Chapter 1	General Introduction to the SoMachine Logic Builder . .	27
	What is the SoMachine Logic Builder?	28
	Tasks Performed by the SoMachine Logic Builder	29
Chapter 2	SoMachine Logic Builder User Interface	31
	Elements of the SoMachine Logic Builder Screen	32
	Multi-Tabbed Navigators	38
	Multi-Tabbed Catalog View	44
	Customizing the User Interface	46
	User Interface in Online Mode	51
	Menus and Commands	52
Chapter 3	Basic Concepts	53
	Introduction and Basic Concepts	53
Part II	Configuration	55
Chapter 4	Installing Devices	57
	Integration of Sercos Devices from Third-Party Vendors	57
Chapter 5	Managing Devices	59
5.1	Adding Devices by Drag and Drop	60
	Adding Devices by Drag and Drop	60
5.2	Adding Devices by Context Menu or Plus Button	63
	Adding a Controller	64
	Adding Expansion Devices	65
	Adding Communication Managers	66
	Adding Devices to a Communication Manager	68
	Adding Devices from Template	70
5.3	Updating Devices	71
	Updating Devices	71
5.4	Converting Devices	73
	Converting Devices	73
5.5	Converting Projects	77
	Converting SoMachine Basic and Twido Projects	77

Chapter 6	Common Device Editor Dialogs	95
6.1	Device Configuration	96
	General Information About Device Editors	97
	Controller Selection	99
	Communication Settings	114
	Configuration	118
	Applications	119
	Files	121
	Log	123
	PLC Settings	126
	Users and Groups	128
	Task Deployment	139
	Status	140
	Information	140
6.2	I/O Mapping	141
	I/O Mapping	142
	Working with the I/O Mapping Dialog	145
	I/O Mapping in Online Mode	148
	Implicit Variables for Forcing I/Os	149
Part III	Program	151
Chapter 7	Program Components	153
7.1	Program Organization Unit (POU)	154
	POU	155
	Adding and Calling POUs	156
	Program	160
	Function	162
	Method	165
	Property	168
	Interface	170
	Interface Property	174
	Action	177
	External Function, Function Block, Method	179
	POUs for Implicit Checks	180
7.2	Function Block	181
	General Information	182
	Function Block Instance	185
	Calling a Function Block	186
	Extension of a Function Block	188

	Implementing Interfaces	191
	Method Invocation	193
	SUPER Pointer	195
	THIS Pointer	197
7.3	Application Objects	200
	Data Type Unit (DUT)	201
	Global Variable List - GVL	203
	Global Network Variable List - GNVL	205
	Persistent Variables	213
	External File	214
	Text List	216
	Image Pool	223
7.4	Application	225
	Application	225
Chapter 8	Task Configuration	227
	Task Configuration	228
	Adding Tasks	229
Chapter 9	Managing Applications	231
9.1	General Information	232
	Introduction	232
9.2	Building and Downloading Applications	234
	Building Applications	235
	Login	236
	Build Process at Changed Applications	238
	Downloading an Application	239
9.3	Running Applications	249
	Running Applications	249
9.4	Maintaining Applications	250
	Monitoring	251
	Debugging	252
Part IV	Logic Editors	255
Chapter 10	FBD/LD/IL Editor	257
10.1	Information on the FBD/LD/IL Editor	258
	FBD/LD/IL Editor	259
	Function Block Diagram (FBD) Language	260
	Ladder Diagram (LD) Language	261
	Instruction List (IL) Language	262
	Modifiers and Operators in IL	264

	Working in the FBD and LD Editor	267
	Working in the IL Editor	272
	Cursor Positions in FBD, LD, and IL	278
	FBD/LD/IL Menu	282
	FBD/LD/IL Editor in Online Mode.	283
10.2	FBD/LD/IL Elements	289
	FBD/LD/IL Toolbox.	290
	Network in FBD/LD/IL	292
	Assignment in FBD/LD/IL	294
	Jump in FBD/LD/IL.	294
	Label in FBD/LD/IL.	294
	Boxes in FBD/LD/IL	295
	RETURN Instruction in FBD/LD/IL	295
	Branch / Hanging Coil in FBD/LD/IL.	296
	Parallel Branch.	299
	Set/Reset in FBD/LD/IL	301
	Set/Reset Coil	302
10.3	LD Elements.	303
	Contact.	304
	Coil	305
Chapter 11	Continuous Function Chart (CFC) Editor	307
	Continuous Function Chart (CFC) Language.	308
	CFC Editor	309
	Cursor Positions in CFC.	311
	CFC Elements / ToolBox	313
	Working in the CFC Editor	319
	CFC Editor in Online Mode	322
	CFC Editor Page-Oriented.	324
Chapter 12	Sequential Function Chart (SFC) Editor.	327
	SFC Editor	328
	SFC - Sequential Function Chart Language	330
	Cursor Positions in SFC.	331
	Working in the SFC Editor	333
	SFC Element Properties	335
	SFC Elements / ToolBox	337

	Qualifier for Actions in SFC	349
	Implicit Variables - SFC Flags	350
	Sequence of Processing in SFC	355
	SFC Editor in Online Mode	357
Chapter 13	Structured Text (ST) Editor	359
13.1	Information on the ST Editor	360
	ST Editor	361
	ST Editor in Online Mode	362
13.2	Structured Text ST / Extended Structured Text (ExST) Language	366
	Structured Text ST / Extended Structured Text ExST	367
	Expressions	368
	Instructions	370
Part V	Object Editors	379
Chapter 14	Declaration Editors	381
	Textual Declaration Editor	382
	Tabular Declaration Editor	383
	Declaration Editor in Online Mode	387
Chapter 15	Device Type Manager (DTM) Editor	389
	DTM Editor	389
Chapter 16	Data Unit Type (DUT) Editor	391
	Data Unit Type Editor	391
Chapter 17	Global Variables List (GVL) Editor	393
	GVL Editor	393
Chapter 18	Network Variables List (NVL) Editor	395
18.1	Information on the NVL Editor	396
	Network Variables List Editor	396
18.2	General Information on Network Variables	397
	Introduction to Network Variables List (NVL)	398
	Configuring the Network Variables Exchange	401
	Network Variables List (NVL) Rules	406
	Operating State of the Sender and the Receiver	408
	Example	409
	Compatibility	415

Chapter 19	Task Editor	419
	Information on the Task Configuration	420
	Properties Tab	421
	Monitor Tab	422
	Configuration of a Specific Task	424
	Task Processing in Online Mode	427
Chapter 20	Watch List Editor	429
	Watch View / Watch List Editor	430
	Creating a Watch List	431
	Watch List in Online Mode	433
Chapter 21	Tools Within Logic Editors	435
	Function and Function Block Finder	436
	Input Assistant	440
Part VI	Tools	443
Chapter 22	Data Logging	445
	Introduction to Data Logging	445
Chapter 23	Recipe Manager	447
	Recipe Manager	448
	Recipe Definition	452
	RecipeMan Commands	456
Chapter 24	Trace Editor	465
24.1	Trace Object	466
	Trace Basics	467
	Creating a Trace Object	469
24.2	Trace Configuration	472
	Variable Settings	473
	Record Settings	477
	Advanced Trace Settings	481
	Edit Appearance	483
	Appearance of the Y-axis	487
24.3	Trace Editor in Online Mode	488
	Trace Editor in Online Mode	488
24.4	Keyboard Operations for Trace Diagrams	489
	Keyboard Shortcuts	489
Chapter 25	Symbol Configuration Editor	491
	Symbol Configuration Editor	492
	Symbol Configuration	495
	Adding a Symbol Configuration	496

Chapter 26	SoMachine Controller - HMI Data Exchange	499
	SoMachine Single Variable Definition	500
	Publishing Variables in the Controller Part	504
	Selecting Variables in the HMI Part	506
	Publishing Variables in the HMI Part	507
	Parametrization of the Physical Media	509
	Communication Performance on Controller - HMI Data Exchange	510
Part VII	Programming Reference	515
Chapter 27	Variables Declaration	517
27.1	Declaration	518
	General Information	519
	Recommendations on the Naming of Identifiers	522
	Variables Initialization	527
	Declaration	528
	Shortcut Mode	529
	AT Declaration	530
	Keywords	531
27.2	Variable Types	535
	Variable Types	536
	Attribute Keywords for Variable Types	539
	Variables Configuration - VAR_CONFIG	543
27.3	Method Types	545
	FB_init, FB_reinit Methods	546
	FB_exit Method	549
27.4	Pragma Instructions	550
	Pragma Instructions	551
	Message Pragmas	553
	Conditional Pragmas	555
27.5	Attribute Pragmas	562
	Attribute Pragmas	563
	User-Defined Attributes	564
	Attribute call_after_init	566
	Attribute displaymode	567
	Attribute ExpandFully	568
	Attribute global_init_slot	569
	Attribute hide	570
	Attribute hide_all_locals	571
	Attribute initialize_on_call	572

Attribute init_namespace.....	573
Attribute init_On_Onlchange.....	574
Attribute instance-path.....	575
Attribute linkalways	576
Attribute monitoring	578
Attribute namespace	582
Attribute no_check	584
Attribute no_copy	584
Attribute no-exit	585
Attribute no_init	586
Attribute no_virtual_actions.....	587
Attribute obsolete	590
Attribute pack_mode	591
Attribute qualified_only.....	592
Attribute reflection	593
Attribute subsequent	593
Attribute symbol.....	594
Attribute warning disable.....	595
27.6 The Smart Coding Functionality.....	596
Smart Coding.....	596
Chapter 28 Data Types.....	599
28.1 General Information	600
Data Types.....	600
28.2 Standard Data Types	601
Standard Data Types	601
28.3 Extensions to IEC Standard	604
UNION	605
LTIME.....	605
WSTRING	606
BIT	606
References.....	607
Pointers	609
28.4 User-Defined Data Types.....	612
Defined Data Types	613
Arrays.....	614
Structures.....	617
Enumerations	619
Subrange Types.....	621

Chapter 29	Programming Guidelines	625
29.1	Naming Conventions	626
	General Information	626
29.2	Prefixes	628
	Prefix Parts	629
	Order of Prefixes	630
	Scope Prefix	632
	Data Type Prefix	633
	Property Prefix	635
	POU Prefix	636
	Namespace Prefix	637
Chapter 30	Operators	639
30.1	Arithmetic Operators	640
	ADD	641
	MUL	643
	SUB	645
	DIV	647
	MOD	650
	MOVE	652
	SIZEOF	653
30.2	Bitstring Operators	654
	AND	655
	OR	656
	XOR	657
	NOT	658
30.3	Bit-Shift Operators	659
	SHL	660
	SHR	662
	ROL	663
	ROR	665
30.4	Selection Operators	667
	SEL	668
	MAX	669
	MIN	670
	LIMIT	671
	MUX	672

30.5	Comparison Operators	673
	GT	674
	LT	675
	LE	676
	GE	677
	EQ	678
	NE	679
30.6	Address Operators	680
	ADR	681
	Content Operator	682
	BITADR	683
30.7	Calling Operator	684
	CAL	684
30.8	Type Conversion Operators	685
	Type Conversion Functions	686
	BOOL_TO Conversions	687
	TO_BOOL Conversions	689
	Conversion Between Integral Number Types	691
	REAL_TO / LREAL_TO Conversions	692
	TIME_TO/TIME_OF_DAY Conversions	694
	DATE_TO/DT_TO Conversions	696
	STRING_TO Conversions	698
	TRUNC	700
	TRUNC_INT	701
	ANY_..._TO Conversions	701
30.9	Numeric Functions	702
	ABS	703
	SQRT	704
	LN	705
	LOG	706
	EXP	707
	SIN	708
	COS	709
	TAN	710
	ASIN	711
	ACOS	712
	ATAN	713
	EXPT	714

30.10	IEC Extending Operators	715
	IEC Extending Operators	716
	__DELETE	717
	__ISVALIDREF	719
	__NEW	720
	__QUERYINTERFACE	723
	__QUERYPOINTER	725
	Scope Operators	727
30.11	Initialization Operator	729
	INI Operator	729
Chapter 31	Operands	731
31.1	Constants	732
	BOOL Constants	733
	TIME Constants	734
	DATE Constants	736
	DATE_AND_TIME Constants	737
	TIME_OF_DAY Constants	738
	Number Constants	739
	REAL/LREAL Constants	740
	STRING Constants	741
	Typed Constants / Typed Literals	742
31.2	Variables	743
	Variables	744
	Addressing Bits in Variables	745
31.3	Addresses	747
	Address	747
31.4	Functions	750
	Functions	750
Part VIII	SoMachine Templates	751
Chapter 32	General Information about SoMachine Templates	753
32.1	SoMachine Templates	754
	General Information About SoMachine Templates	755
	Administration of SoMachine Templates	758

Chapter 33	Managing Device Templates	765
33.1	Managing Device Templates	766
	Facts of Device Templates	767
	Adding Devices from Template	768
	Creating a Device Template on the Basis of Field Devices or I/O Modules	771
	Visualizations Suitable for Creating Device Templates	772
	Further Information on Integrating Control Logic into Device Templates	773
	Steps to Create a Device Template	775
Chapter 34	Managing Function Templates	779
34.1	Managing Function Templates	780
	Facts of Function Templates	781
	Adding Functions from Template	782
	Application Functions as Basis for Function Templates	788
	Steps to Create a Function Template	790
Part IX	Troubleshooting and FAQ	795
Chapter 35	Generic - Troubleshooting and FAQ	797
35.1	Frequently Asked Questions	798
	How Can I Enable and Configure Analog Inputs on CANopen?	799
	Why is SoMachine Startup Performance Sometimes Slower?	800
	How Can I Manage Shortcuts and Menus?	801
	How Can I Increase the Memory Limit Available for SoMachine on 32-Bit Operating Systems?	803
	How Can I Reduce the Memory Consumption of SoMachine?	804
	How Can I Increase the Build-Time Performance of SoMachine?	804
	What Can I Do in Case of Issues with Modbus IOScanner on Serial Line?	805
	What Can I Do If My Network Variables List (NVL) Communication Has Been Suspended?	806
	What Can I Do If a Multiple Download is Unsuccessful on an HMI Controller?	806
Chapter 36	Accessing Controllers - Troubleshooting and FAQ	807
36.1	Troubleshooting: Accessing New Controllers	808
	Accessing New Controllers	809
	Connecting via IP Address and Address Information	811
36.2	FAQ - What Can I Do in Case of Connection Problems With the Controller?	813
	FAQ - Why is a Connection to the Controller not Possible?	814
	FAQ - Why has the Communication Between PC and Controller been Interrupted?	817

Appendices	819
Appendix A Network Communication	821
Network Topology	822
Addressing and Routing	823
Structure of Addresses	825
Appendix B Usage of the OPC Server 3	829
General Information	830
Declaring a Variable to be Used With OPC	832
OPC Server Configuration	836
Usage of the CoDeSys OPC Server	842
Appendix C Script Language	843
C.1 General Information	844
Introduction	845
Executing Scripts	848
Best Practices	850
Reading .NET API Documentations	851
Entry Points	853
C.2 Schneider Electric Script Engine Examples	854
Device Parameters	855
Compiler Version	857
Visualization Profile	858
Update Libraries	859
Clean and Build Application	860
Communication Settings	861
Reset Diagnostic Messages	862
Reboot the Controller	862
Convert Device	863
Comparing Projects	866
Advanced Library Management Functions	867
Accessing POUs	868
C.3 CoDeSys Script Engine Examples	869
Project	870
Online Application	875
Objects	878
Devices	879
System / User Interface (UI)	881

	Reading Values	883
	Reading Values From Recipe and Send an Email.....	884
	Determine Device Tree of the Open Project	886
	Script Example 4: Import a Device in PLCOpenXML From Subversion	887
Appendix D	User Management for Soft PLC	889
	General Information on User Management for Soft PLC	890
	Users and Groups	892
	Access Rights	895
Appendix E	Controller Feature Sets for Migration	899
	Controller Feature Sets for Migration	899
Glossary		903
Index		907

Safety Information



Important Information

NOTICE

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 Book



At a Glance

Document Scope

This document describes the graphical user interface of the SoMachine software and the functions it provides. For further information, refer to the separate documents provided in the SoMachine online help.

Validity Note

This document has been updated for the release of TM3TI4D Add-on for SoMachine V4.3.

Related Documents

Document title	Reference
SoMachine Introduction	<i>EIO0000000734 (ENG);</i> <i>EIO0000000787 (FRE);</i> <i>EIO0000000788 (GER);</i> <i>EIO0000000790 (SPA);</i> <i>EIO0000000789 (ITA);</i> <i>EIO0000000791 (CHS)</i>
SoMachine Menu Commands Online Help	EIO0000001854 (ENG); EIO0000001855 (FRE); EIO0000001856 (GER); EIO0000001858 (SPA); EIO0000001857 (ITA); EIO0000001859 (CHS)
SoMachine Central User Guide	<i>EIO0000001659 (ENG);</i> <i>EIO0000001660 (FRE);</i> <i>EIO0000001661 (GER);</i> <i>EIO0000001663 (SPA);</i> <i>EIO0000001662 (ITA);</i> <i>EIO0000001664 (CHS)</i>
SoMachine Compatibility and Migration User Guide	<i>EIO0000001684 (ENG);</i> <i>EIO0000001685 (FRE);</i> <i>EIO0000001686 (GER);</i> <i>EIO0000001688 (SPA);</i> <i>EIO0000001687 (ITA);</i> <i>EIO0000001689 (CHS)</i>

Document title	Reference
SoMachine Functions and Libraries User Guide	<i>EIO0000000735 (ENG);</i> <i>EIO0000000792 (FRE);</i> <i>EIO0000000793 (GER);</i> <i>EIO0000000795 (SPA);</i> <i>EIO0000000794 (ITA);</i> <i>EIO0000000796 (CHS)</i>
SoMachine Controller Assistant User Guide	<i>EIO0000001671 (ENG);</i> <i>EIO0000001672 (FRE);</i> <i>EIO0000001673 (GER);</i> <i>EIO0000001675 (SPA);</i> <i>EIO0000001674 (ITA);</i> <i>EIO0000001676 (CHS)</i>
Modicon M238 Logic Controller Programming Guide	<i>EIO0000000384 (ENG);</i> <i>EIO0000000385 (FRE);</i> <i>EIO0000000386 (GER);</i> <i>EIO0000000388 (SPA);</i> <i>EIO0000000387 (ITA);</i> <i>EIO0000000389 (CHS)</i>
SoMachine Device Type Manager (DTM) User Guide	<i>EIO0000000673 (ENG);</i> <i>EIO0000000674 (FRE);</i> <i>EIO0000000675 (GER);</i> <i>EIO0000000676 (SPA);</i> <i>EIO0000000677 (ITA);</i> <i>EIO0000000678 (CHS)</i>
TwidoEmulationSupport Library Guide	<i>EIO0000001692 (ENG);</i> <i>EIO0000001693 (FRE);</i> <i>EIO0000001694 (GER);</i> <i>EIO0000001696 (SPA);</i> <i>EIO0000001695 (ITA);</i> <i>EIO0000001697 (CHS)</i>
SoMachine Network Variable Configuration SE_NetVarUdp Library Guide	<i>EIO0000001151 (ENG);</i> <i>EIO0000001152 (FRE);</i> <i>EIO0000001153 (GER);</i> <i>EIO0000001155 (SPA);</i> <i>EIO0000001154 (ITA);</i> <i>EIO0000001156 (CHS)</i>

You can download these technical publications and other technical information from our website at <http://www.schneider-electric.com/en/download>.

Product Related Information

WARNING

LOSS OF CONTROL

- The designer of any control scheme must consider the potential failure modes of control paths and, for certain critical control functions, provide a means to achieve a safe state during and after a path failure. Examples of critical control functions are emergency stop and overtravel stop, power outage and restart.
- Separate or redundant control paths must be provided for critical control functions.
- System control paths may include communication links. Consideration must be given to the implications of unanticipated transmission delays or failures of the link.
- Observe all accident prevention regulations and local safety guidelines.¹
- Each implementation of this equipment must be individually and thoroughly tested for proper operation before being placed 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.

Terminology Derived from Standards

The technical terms, terminology, symbols and the corresponding descriptions in this manual, 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
EN 61131-2:2007	Programmable controllers, part 2: Equipment requirements and tests.
ISO 13849-1:2008	Safety of machinery: Safety related parts of control systems. General principles for design.
EN 61496-1:2013	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
EN 1088:2008 ISO 14119:2013	Safety of machinery - Interlocking devices associated with guards - Principles for design and selection
ISO 13850:2006	Safety of machinery - Emergency stop - Principles for design
EN/IEC 62061:2005	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 61784-3:2008	Digital data communication for measurement and control: Functional safety field buses.
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

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.

Part I

Introduction

What Is in This Part?

This part contains the following chapters:

Chapter	Chapter Name	Page
1	General Introduction to the SoMachine Logic Builder	27
2	SoMachine Logic Builder User Interface	31
3	Basic Concepts	53

Chapter 1

General Introduction to the SoMachine Logic Builder

What Is in This Chapter?

This chapter contains the following topics:

Topic	Page
What is the SoMachine Logic Builder?	28
Tasks Performed by the SoMachine Logic Builder	29

What is the SoMachine Logic Builder?

General Description

The Logic Builder provides the configuration and programming environment for the SoMachine projects you create with SoMachine Central.

It displays the different elements of your project in separate views that you can arrange on the SoMachine 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 SoMachine Logic Builder

Configuring and Programming Projects

The Logic Builder allows you to program logic and add devices to the SoMachine projects you create with SoMachine Central.

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 simple drag and drop. It also allows you to use device templates and function templates.
- Separate software catalog views for **Variables**, **Assets**, **Macros**, **ToolBox**, **Libraries** allow you to add different types of software elements by simple drag and drop. The **Assets** view, for example, allows you to create and manage your function blocks and POU's.

To display only the relevant views for the task that is being performed, SoMachine provides individual perspectives (*see page 49*) 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 **Build**, **Build all**, or **Clean all**) to build your SoMachine 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 and Monitoring

The Logic Builder online and monitoring 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

Chapter 2

SoMachine Logic Builder User Interface

What Is in This Chapter?

This chapter contains the following topics:

Topic	Page
Elements of the SoMachine Logic Builder Screen	32
Multi-Tabbed Navigators	38
Multi-Tabbed Catalog View	44
Customizing the User Interface	46
User Interface in Online Mode	51
Menus and Commands	52

Elements of the SoMachine Logic Builder Screen

Overview

Logic Builder consists of the following elements:

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

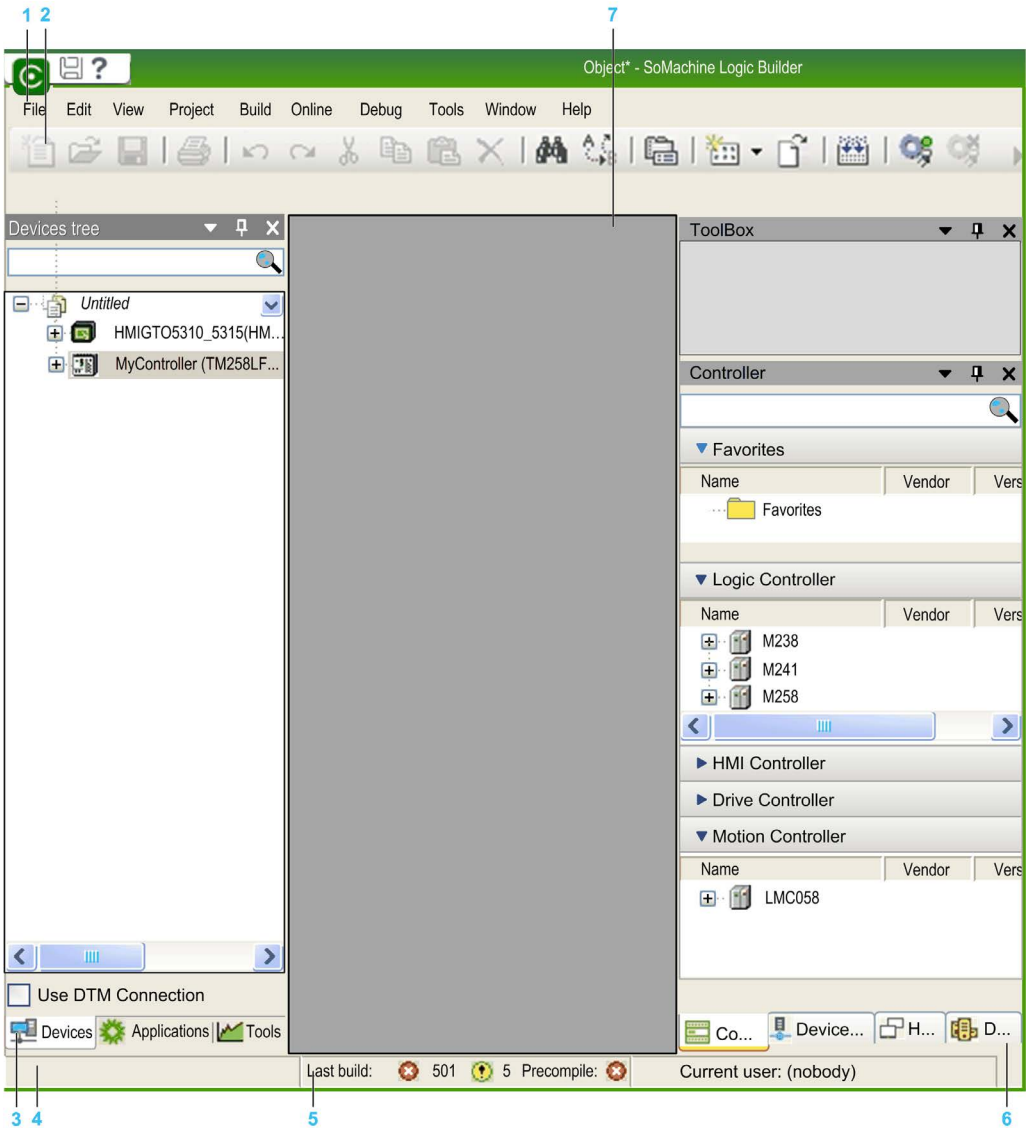
When you open the Logic Builder, it provides a default screen layout. This document describes the default positions.

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

You can also arrange the views and windows anytime via 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 ([see page 49](#)).

Default Logic Builder Screen

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



- 1 Menu bar
- 2 Toolbar
- 3 Multi-tabbed **Navigators**: **Devices tree**, **Tools tree**, **Applications tree**
- 4 **Messages** view
- 5 Information and status bar
- 6 Multi-tabbed catalog view: hardware catalog: **Controller**, **HMI & iPC**, **Devices & Modules**, **Diverse** software catalog: **Variables**, **Assets**, **Macros**, **ToolBox**, **Libraries**
- 7 Multi-tabbed editor view

Standard 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.
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 For further information, refer to the chapter <i>Multi-Tabbed Navigators</i> (see page 38).
Messages view	Provides messages on precompile, compile, build, download operations. Refer to the description of the Messages view commands for details (<i>see SoMachine, Menu Commands, Online Help</i>).
Information and status bar	Provides the following information: • Information on the current user. • Information on editing mode and current position if an editor is open. For further information, see the <i>Information and Status Bar</i> section in this chapter.
Multi-tabbed Catalog view	The Catalog view consists of different tabs where the available hardware and software objects are listed: • Hardware Catalog ○ Controller ○ HMI & iPC ○ Devices & Modules ○ Diverse • Software Catalog ○ Variables ○ Assets ○ Macros ○ ToolBox ○ Libraries For further information, refer to the chapter <i>Multi-Tabbed Catalog Views</i> (see page 44).

Component	Description
Multi-tabbed editor window	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 window in offline or online mode by executing the Edit Object command.

Information and Status Bar

The bar at the lower border of the Logic Builder screen provides 3 types of information:

- Information on the logged-in user.
- If you are working in an editor window: the position of the cursor and the status of editing mode.
- In online mode: the current status of the program.

Information on the logged-in user

Each project has a user and access management (*see page 128*). The currently logged-in user is named in the status bar.

Cursor positions in editor windows

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

Abbreviation	Description
Ln	Line in which the cursor is placed.
Col	Column in which the cursor is placed. (A column includes exactly 1 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, 4 columns.)

Double-click one of the fields 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 following status of the program is indicated:

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

Watch Windows and Online Views of Editors

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

Windows, Views, and Editor Windows

There are 2 different types of windows in the Logic Builder:

- Some can be docked to any margin of the SoMachine window or can be positioned on the screen as undocked windows independently from the SoMachine window. Additionally they can be hidden by being represented as a tab in the SoMachine window frame (refer to the *Customizing the User Interface* chapter ([see 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 via the **View** menu (*see SoMachine, 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 window. You cannot hide or undock them from the SoMachine window. You can access them via the **Window** menu.

Switching Windows

SoMachine allows you to switch between open views and editors. To switch between open views and editors, press the CTRL and TAB keys simultaneously. A window opens that lists the views and editors that are currently 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 standard 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.

You can access views via 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**, simply 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 executing the command **Add Object** or **Add Device**. The device type which can be inserted depends on the currently 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 current position in the tree are available for insertion.

Repositioning Objects

To reposition objects, use the standard 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 (*see page 71*)
- **Convert Device** command (*see page 73*)

Description of the Devices Tree

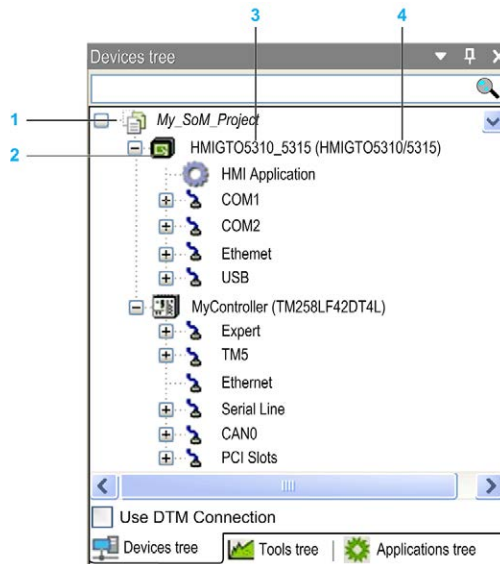
Each device object in the **Devices tree** represents a specific (target) hardware object.

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* (*see page 228*). 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 SoMachine Device Type Manager (DTM) User Guide (*see SoMachine, Device Type Manager (DTM), User Guide*).
- Consider the recommendations for *Adding Elements to the Navigators* in this chapter.

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** shows the symbol, the symbolic name (editable), and the device type (= device name as provided by the device description).
- 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 (device editor). To open the device editor, double-click the device node in the **Devices tree** (refer to the description of the device editor ([see page 97](#))).
- 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 ([see page 126](#)) 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 ([see page 126](#)) is disabled.
 -  Controller is connected, active application is running, diagnostic information is available.

-  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 currently 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 *SoMachine, Menu Commands, Online Help*).
- The names of all currently 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 (see page 140).

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 *SoMachine, Menu Commands, Online Help*), an entry in the **Devices tree** is displayed in italics, and you can log into the application.

You can also connect to the controller in online configuration mode (refer to chapter *Online Config Mode* (see *SoMachine, Menu Commands, Online Help*)) without the need of first having loaded a real application into the controller. This is useful for the initial start-up of an I/O system because you can access and test the I/Os in the controller configuration before you build and load a real application program.

For information on the conversion of device references when opening projects, refer to the *SoMachine 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**, simply 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 executing the command **Add Object** or **Add Device**. The device type which can be inserted depends on the currently 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 current position in the tree are available for insertion.

Repositioning objects:

To reposition objects, use the standard 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 currently 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, this 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 a programmable device again.

Inserting devices below a device object:

You can insert further devices below a device object. If they are installed on the local system and thus 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 – each 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 of the device editor.

Programming objects, like particular POU's 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 all devices.
- When they are declared as a subnode of the **Applications** node, these objects can only be accessed by the corresponding 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 respective program calls have to be defined (instances of POU's from the **Global** node of the **Applications tree** or device-specific POU's). Consider that the application is defined in the **I/O Mapping** view of the respective device editor (*see page 142*).

Description of the Tools Tree

Libraries are managed in the **Tools tree**. Pure configurable devices cannot be assigned such programming objects. You can edit the values of the device parameters in the parameter dialog 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 all devices.
- When they are declared as a subnode of the **Applications** node; then these objects can only be accessed by the corresponding devices declared in this **Applications** node.

Multi-Tabbed Catalog View

Overview

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

It contains the following tabs:

- **Controller:** Contains the **Logic**, **HMI**, **Drive**, and **Motion** controllers that can be inserted in your SoMachine project.
- **Devices & Modules:** Contains the **I/O Modules**, and the **Communication**, **Motor Control**, **Safety**, and **Energy Management** devices that can be inserted in your SoMachine 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 SoMachine project.
- **Diverse:** Contains third party devices that can be inserted in your SoMachine project.

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

You can extend this view by the tabs of the **Software Catalog** (**Variables**, **Assets**, **Macros**, **ToolBox**, **Libraries**) 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 simple drag and drop as described in the *Adding Devices by Drag and Drop* chapter ([see page 60](#)).

Searching Within Catalogs

Each tab of the catalog view contains a search box. The sub-lists of the tab are searched for the string you enter in the search box. In open sub-lists, the found entires 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 sub-lists is displayed in bold print in the title bar of each the sub-list.

By default, the search is executed on the names of the items in the lists. But SoMachine also supports the tagging mechanism. 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 (*see page 768*).

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:

- Standard pre-settings for menus, keyboard functions, and toolbars. You can overwrite the SoMachine default settings via the **Customize** dialog box (*see SoMachine, Menu Commands, Online Help*) (by default available in the **Tools** menu). The current 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 SoMachine, Menu Commands, Online Help*). You can also overwrite these settings. The current configuration is saved on the local system.
- The way you arrange views or editor windows within the project. The current 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 (*see 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: By default, you can reopen the views of standard components via the **View** menu. To open an editor window, execute the command **Project → Edit object** or double-click the respective entry in the **Devices tree**, **Applications tree**, or in the **Tools tree**.

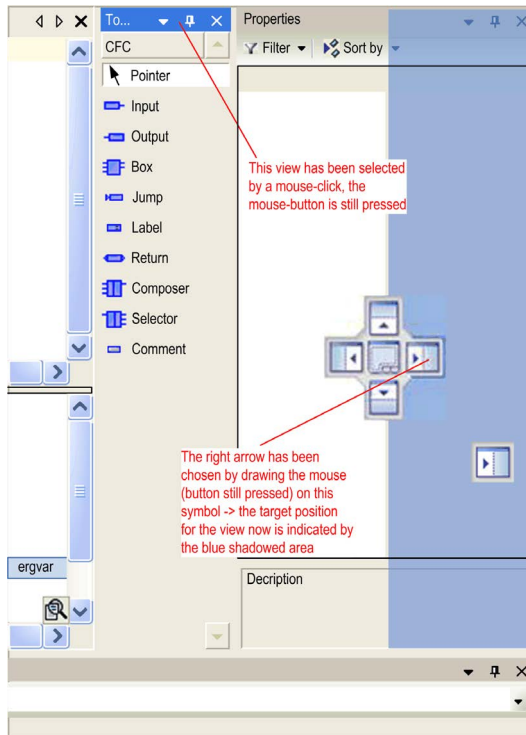
Resizing a view or window within the frame window: Move the separator lines between neighboring views. You can resize independent view windows on the desktop by moving the window borders.

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 alternatively the tab of the view, keep the mouse-button pressed, and move the view to the desired place. Arrow symbols will display showing every possible target position. Keep the mouse-button pressed and choose the desired position by moving the cursor on the respective arrow symbol. 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 loses the **Auto Hide** button. As an alternative, execute the commands **Dock** and **Float** from the **Window** menu.

Hiding views: You can hide views with **Auto Hide** buttons at the border of the SoMachine 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 left**.

Unhiding views: To unhide a view, click the **Auto Hide** left 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 (*see page 33*).

Perspectives

A perspective is used to save the layout of SoMachine 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, SoMachine provides 4 perspectives for the following use cases in the **Window → Switch Perspective** menu or in the perspective table in the toolbar.

Perspective name	Use case	Navigators (on the left side)	Catalog views (on the right side)	Views at the bottom of the screen
Device Configuration	For adding / configuring devices.	• Devices tree • Applications tree • Tools tree	Hardware catalog • Controller • Devices & Modules • HMI & iPC • Diverse	Messages (in Auto Hide mode)
Logic Configuration	For adding / creating logic.	• Devices tree • Applications tree • Tools tree	Software catalog • Variables • Assets • Macros • ToolBox • Libraries	Messages (in Auto Hide mode)
CODESYS Classic	Standard CoDeSys views.	• Devices • POUs	Hardware catalog • Controller • Devices & Modules • HMI & iPC • Diverse	Messages (in Auto Hide mode)
Online	For online mode.	• Devices tree • Applications tree • Tools tree	Hardware catalog • Controller • Devices & Modules • HMI & iPC • Diverse	• Messages (in Auto Hide mode) • Watch 1

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

Creating your own perspective:

In addition to these standard 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	Execute the command Save current view layout as perspective from the Window menu to save your modifications to a new perspective.
3	In the Save current view layout as perspective dialog box, enter a Name for your perspective. Result: The current 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, execute the command **Reset current Perspective** from the **Window** menu.

Importing / exporting perspectives:

To be able to exchange perspectives between different SoMachine installations or between different users, the **Tools → Options → Perspectives** dialog box (*see SoMachine, 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

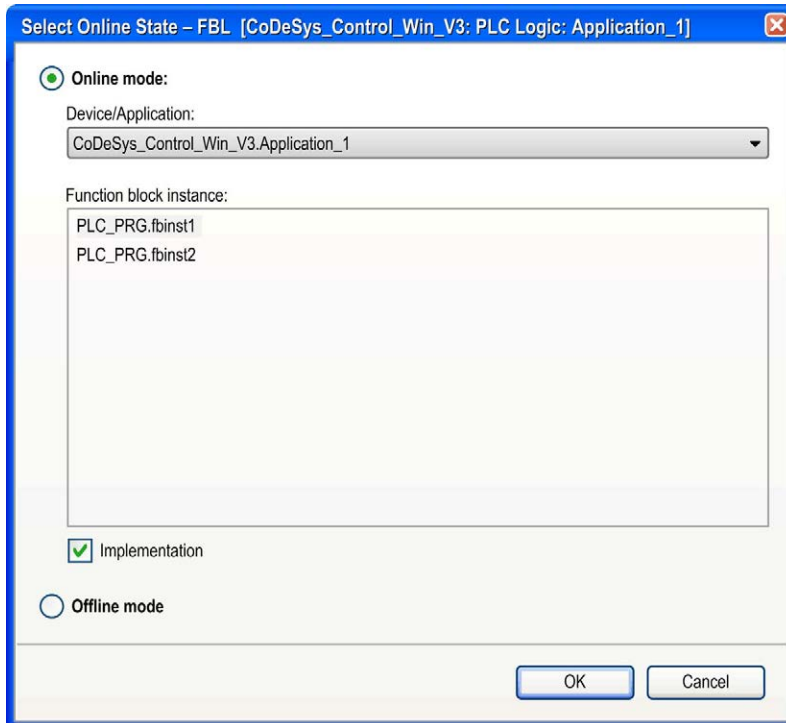
Overview

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 (*see page 49*) which means that the **Watch** view opens by default.

To open an object in online mode which is not already open, double-click the node in the **Applications tree** or execute the **Project → Edit Object** command. The object will be opened in online mode.

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>** will display. 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 respective 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 (*see page 33*) 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, insert it in a menu of your choice by using the **Tools → Customize** dialog box (*see SoMachine, Menu Commands, Online Help*).

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.

To reorganize the menu structures, use the **Tools → Customize** dialog box.

Several commands of the **File** menu are not available because these tasks are performed in the SoMachine Central. For further information, refer to the SoMachine Central User Guide (*see SoMachine Central, User Guide*).

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

Chapter 3

Basic Concepts

Introduction and Basic Concepts

Overview

SoMachine is a device-independent controller programming system.

Conforming to the IEC 61131-3 standard, it supports all standard programming languages. Further, it allows including C-routines. It allows you to program multiple controller devices within one project.

For further information see chapter ***Generate runtime system files*** (see *SoMachine, Menu Commands, Online Help*).

Object Orientation

The object-oriented approach is not only reflected by the availability of appropriate programming elements and features, but also in the structure and version handling of SoMachine and in the project organization. Multi-device usage of a SoMachine project is possible based on jointly used, instantiated programming units. Cloning of applications is possible as well as mixing configurable and programmable controller devices in one project.

Version Handling

A simultaneous installation of several versions of SoMachine components and working with the desired combination of versions is possible. This also pertains 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 *SoMachine Compatibility and Migration User Guide*.

Project Organization

Project organization is also determined in an object oriented fashion. A SoMachine 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) (<i>see page 155</i>):	These are programs, functions, function blocks, methods, interfaces, actions, data type, definitions, and so on. Programming objects which can be instantiated in the entire project, that is, for all applications defined in the project, must be managed in the Global node of the Applications tree . The instantiating is done by calling a program POU by an application-assigned task. Programming objects which are only managed in the Applications tree , that is, which are directly assigned to an application, cannot only be instantiated by another application inserted below.
Resource objects (Devices tree)	Device objects are only managed in the Devices tree . When you insert objects in the Devices tree , consider the recommendations described in the <i>Adding Elements to the Navigators</i> section (<i>see page 41</i>).

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 SoMachine and the device is conducted via a gateway (component) and a runtime system. Complete online functionality for controlling a program on the device is available.

Supported Programming Languages

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

- FBD/LD/IL editor (*see page 257*) for function block diagram (FBD), ladder logic diagram (LD), and instruction list (IL)
- SFC editor (*see page 327*) for sequential function chart
- ST editor (*see page 359*) for structured text

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

- CFC editor (*see page 307*) for continuous function chart

CFC is an extension to the standard IEC programming languages.

Part II

Configuration

What Is in This Part?

This part contains the following chapters:

Chapter	Chapter Name	Page
4	Installing Devices	57
5	Managing Devices	59
6	Common Device Editor Dialogs	95

Chapter 4

Installing Devices

Integration of Sercos Devices from Third-Party Vendors

Introduction

Via the **Device Repository** dialog box (*see SoMachine, Menu Commands, Online Help*), you can 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 SoMachine

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 for SoMachine.

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 SoMachine, 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 *SoMachine, Menu Commands, Online Help*).

Chapter 5

Managing Devices

What Is in This Chapter?

This chapter contains the following sections:

Section	Topic	Page
5.1	Adding Devices by Drag and Drop	60
5.2	Adding Devices by Context Menu or Plus Button	63
5.3	Updating Devices	71
5.4	Converting Devices	73
5.5	Converting Projects	77

Section 5.1

Adding Devices by Drag and Drop

Adding Devices by Drag and Drop

Overview

SoMachine V4.0 and later versions provide a multi-tabbed catalog view on the right-hand side of the SoMachine Logic Builder.

2 different types of catalog views are available:

- The **Hardware Catalog**
- The **Software Catalog**

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 by Drag and Drop

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 SoMachine Logic Builder toolbar if it is not already opened.
2	Select the tab Controller in the Hardware Catalog . Result: The controllers suitable for your SoMachine 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 by Drag and Drop

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 SoMachine Logic Builder toolbar if it is not already opened.
2	Select the tab Devices & Modules in the Hardware Catalog . Result: The expansion devices suitable for your SoMachine project are displayed in the Hardware Catalog .
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 by SoMachine. 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 by Drag and Drop

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 SoMachine Logic Builder toolbar if it is not already opened.
2	Select the tab Devices & Modules in the Hardware Catalog . Result: The field devices suitable for your SoMachine 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 by SoMachine. 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 by Drag and Drop

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 SoMachine Logic Builder 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 SoMachine project are displayed in the Devices & Modules tab.
4	Add them to the Devices tree as described in the <i>Adding Devices from Template</i> chapter (<i>see page 768</i>).

Adding Devices from Function Template by Drag and Drop

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

Step	Action
1	Open the software catalog by clicking the Software Catalog button  in the SoMachine Logic Builder toolbar if it is not already opened.
2	Select the tab Macro in the Software Catalog . Result: The function templates available in SoMachine are displayed in the Software Catalog .
3	Select a function template entry in the Macro view, drag it to the Devices tree , and drop it at a suitable subnode of a controller. NOTE: Suitable subnodes are expanded and highlighted by SoMachine. Result: The device based on the function template is added to the Devices tree .

Section 5.2

Adding Devices by Context Menu or Plus Button

What Is in This Section?

This section contains the following topics:

Topic	Page
Adding a Controller	64
Adding Expansion Devices	65
Adding Communication Managers	66
Adding Devices to a Communication Manager	68
Adding Devices from Template	70

Adding a Controller

Introduction

As an alternative to dragging and dropping devices on the **Devices tree**, 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 add a suitable device using the context 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 *SoMachine, 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 device to your SoMachine project, proceed as follows:

Step	Action
1	Select a project node, click the green plus button of the node, or right-click the project node and select the Add Device... command from the context 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. • Or 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 *SoMachine 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 select the Add Device... command from the context 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. • Or you can click the Close button.

NOTE: When you add a TWDNOI10M3 object (AS-Interface Master Module), the corresponding **Virtual AS interface bus** fieldbus manager will automatically be inserted. For further information on AS interface configuration, refer to the chapter in the Modicon M238 Logic Controller Programming Guide (*see Modicon M238 Logic Controller, Programming Guide*).

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.

2 types of communication managers exist:

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

Communication managers available in SoMachine are listed below:

Name	Interface type	Description
ASCII Manager	Serial line	Used to transmit and/or receive data with a simple device.
SoMachine-Network Manager	• Serial line (max. 1) • Ethernet (max. 3)	Use it if you want to connect an XBTGT, XBTGK, XBTGH or SCU HMI through SoMachine protocol offering transparent exchange of data and multiple download capability (download of controller and HMI applications through 1 unique connection PC-controller or PC-HMI). A maximum of 4 connections is available: 1 for SoMachine (even if a USB connection is used), 3 for Ethernet.
Modbus IOScanner	Serial line	Modbus RTU or ASCII protocol manager used to define implicit exchanges (I/O scanning) with Modbus slave devices.
Modbus Manager	Serial line	Used for Modbus RTU or ASCII protocol in master or slave mode.
CANopen Optimized	CAN	CANopen manager for optimized controllers (M238, M241, XBTGC, XBTGT, XBTGK, SCU HMI, ATV IMC)
CANopen Performance	CAN	CANopen manager for performance controllers (M251, M258 and LMC058 and LMC078)
CANmotion	CAN	CANmotion manager for LMC058 and LMC078 Motion Controller CAN1 port only.
Modbus TCP Slave Device	Ethernet	Modbus TCP manager for controllers with Ethernet port.
EtherNet/IP	Ethernet	EtherNet/IP manager for controllers with Ethernet port (M251, M258, LMC058 and LMC078).

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 , CAN0 , CAN1 , 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 context 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 sort the devices by brand by clicking the list box Vendor .
3	Select the Communication manager from the list below.
4	Rename your device by typing a name in the Name textbox. 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 execute the **Add Device** command.

As a prerequisite, the device must be available in the **Device Repository** dialog box (see *SoMachine, 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 select the Add Device... command from the context 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 sort 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, for whatever reason.

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

- Monitoring the response of the slaves configured via SDO messages.
- Monitoring the master for any 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].udiAb
ortCode;
  (* 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 ([see page 768](#)).

Section 5.3

Updating Devices

Updating Devices

Introduction

The update device function allows you to replace a device selected in the **Devices tree**

- by another version of the same device or
- by a different type of device.

Updating Devices

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

Step	Action
1	Select the device you want to replace in the Devices tree and execute the command Update Device... from the Project menu. OR Right-click the device you want to replace in the Devices tree and select the command Update Device... from the context menu. Result: The Update Device dialog box opens. OR Right-click the device you want to replace in the Devices tree and select the command Add Device... from the context 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 device that should replace the current device. To select a specific version of the device, select the options Display all versions (for experts only) and/or Display outdated versions.
3	If necessary to distinguish the devices, rename your device by typing a name in the text box Name. Otherwise, the same name will be used for the updated device. If the device is updated by a different device type, then the description of the device type (in brackets behind the device name) will be automatically adapted. Naming the device meaningfully may ease the organization of your project.
4	Click the Update Device button. Result: The device that had been selected in the Devices tree is replaced by the new device type or the new version. The new device type or the new 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 appropriate messages. This also concerns implicitly added libraries which will not be removed automatically and appropriately at a device update.

Section 5.4

Converting Devices

Converting Devices

Introduction

SoMachine 4.0 and later versions allow you to convert a device that is configured in your project to a different, but compatible device. SoMachine automatically converts the currently configured device into the selected device and displays the changes that are made 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 was 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 hard disk of the PC before converting a device.

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 hard disk of 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 hard disk of 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	Select the Convert Device command from the context menu. Result: The Convert Device dialog box is displayed. It lists those devices that are compatible to the device you selected and provides further information on the selected device:

Convert device

Selected devices

Name: TM238LDD24DT
Type: TM238LFAC24DR
Version: 2.0.40.6

Compatible devices

Manufacturer: Schneider Electric

Name	Manufacturer	Version
Logic Controller		
M238		
TM238LFDC24DT	Schneider Electric	2.0.40.6
TM238LDA24DR	Schneider Electric	2.0.40.6
TM238LDD24DT	Schneider Electric	2.0.40.6
M258		
TM258LD42DT	Schneider Electric	2.0.40.11
TM258LD42DT4L	Schneider Electric	2.0.40.11
TM258LF42DT	Schneider Electric	2.0.40.11
TM258LF42DT4L	Schneider Electric	2.0.40.11

☐ Display all versions (for experts only)

Information

Name: TM238LDA24DR
Manufacturer: Schneider Electric
Groups: PLCs
Version: 2.0.40.6
Model number: TM238LDA24DR
Description: Compact base controller – AC Power Supply, 14 normal inputs, 6 Relay outputs, 4 transistor outputs (0.5 A). Timer and calendar. Removable terminal blocks.



OK Cancel

Step	Action
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	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 conserved if the related modules are still available. Any modifications or configurations that could not be converted are listed in the Messages view.
6	Check 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:

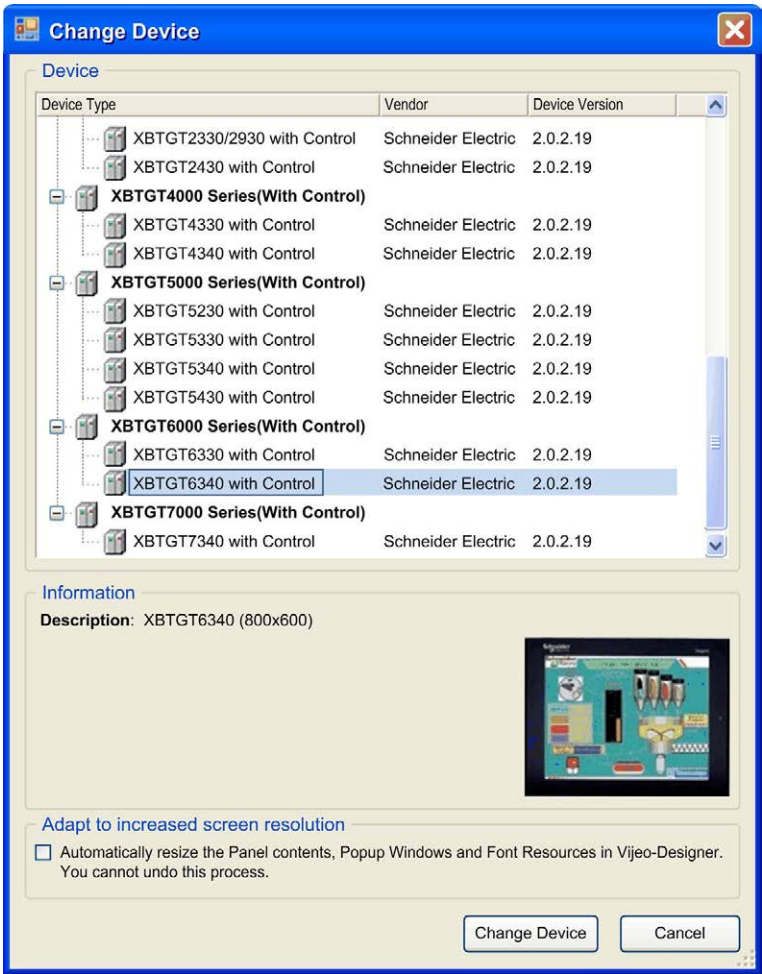
- the source devices and the target devices they have been converted to
- the parameters that have not been transferred to the target
- the devices that have not been converted

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).

Special Case: Converting an HMI Device to Another HMI Device with Higher Screen Resolution

Like the other devices, you can also convert an HMI device to another HMI device. In this case, the **Convert Device** dialog box includes an additional option for HMI devices that allows automatic adaptation to a higher screen resolution.

Convert Device dialog box for HMI devices



If the new HMI device has a bigger screen and thus a higher screen resolution, the option **Adapt to increased screen resolution** is by default enabled. It automatically adapts the contents of the HMI panels and the popup windows as well as the fonts of the HMI panels to the increased screen resolution of the new HMI device.

NOTE: This process cannot be undone automatically. Verify and, if necessary, manually adapt the contents of the panels after the conversion.

Section 5.5

Converting Projects

Converting SoMachine Basic and Twido Projects

Introduction

With SoMachine, you can convert a SoMachine Basic or TwidoSoft/TwidoSuite project and the configured controller to a selectable SoMachine logic or HMI controller (*see page 899*). The controller and the corresponding logic are converted and integrated in the SoMachine project.

SoMachine provides different ways to execute this conversion process:

- In SoMachine Central, execute the command **Convert... → Convert Twido Project...** or **Convert... → Convert SoMachine Basic Project...** from the Main Menu (*see SoMachine Central, User Guide*). The **Convert SoMachine Basic Project** dialog box or **Convert Twido Project** dialog box opens.
- In SoMachine Logic Builder, 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 SoMachine, Menu Commands, Online Help*).
- In SoMachine Central, open a SoMachine Basic or Twido project. To achieve this, click the **Open an existing project** button, or click **Open Project** in the **Get started** screen to open the **Open project** dialog box. Select the option **Twido Project files (*.xpr, *.twd, *.xar)** or **SoMachine Basic Project files (*.smbp)** from the list of **All supported files** and browse for your SoMachine Basic or Twido project. Click the **Open** button to open the **Convert SoMachine Basic Project** dialog box or the **Convert Twido Project** dialog box.

The SoMachine Basic versions supported by this conversion mechanism are listed in the release notes of SoMachine. 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 SoMachine, 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 SoMachine.

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

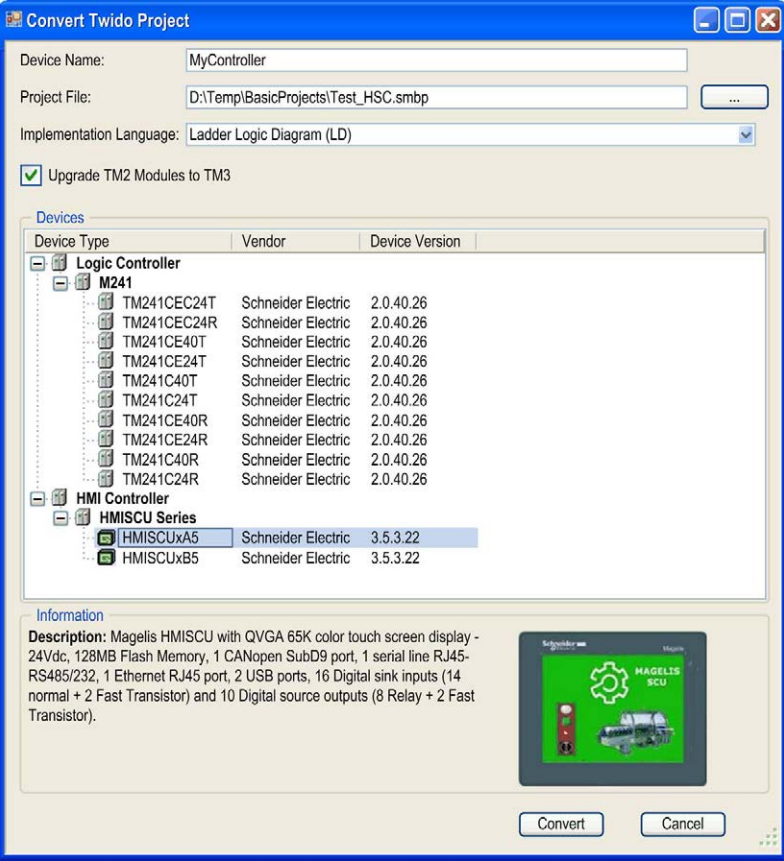
- Verify that the program for the target controller contains the intended configurations and provides the intended functions after you have converted the project.
- Fully debug, verify, and validate the functionality of the converted program before putting it into service.
- Before converting a program, 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 SoMachine, see the *SoMachine Compatibility and Migration User Guide* (see *SoMachine 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 SoMachine Central or the SoMachine Logic Builder (as listed in the <i>Introduction</i> block of this chapter (see page 77)). 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.

Step	Action
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 SoMachine project. Modifications or configurations that could not be converted are listed in the Messages view (see <i>SoMachine, Menu Commands, Online Help</i>).
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 SoMachine 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 SoMachine project, activate the option **Allow unicode characters for identifiers** in the **Project Settings → Compile options** dialog box (see *SoMachine, 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 a SoMachine application. The TwidoEmulationSupport library is automatically integrated in the SoMachine project with the converted controller.

Conversion of the Application Program

In the target SoMachine 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 Grafset. They are converted to an SFC program. For detailed information, refer to the Grafset section in this chapter (see page 88).

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 (*see page 203*) 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 SoMachine, it is not possible to jump to a label (*see page 294*) in another program.
- It is not possible to define Grafset steps in a subprogram.
- It is not possible to activate or deactivate Grafset 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 SoMachine.

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 SoMachine, 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 SoMachine.



- 1 Memory addresses in SoMachine Basic / Twido
- 2 Memory addresses in SoMachine

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 SoMachine application.
- If a remote device accesses %M objects in the source application, this access will no longer be available in the SoMachine 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 <i>Conversion of Fast Counters, High-speed Counters (Twido: Very Fast Counters) and Pulse Generators (see page 86)</i> 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	

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* ([see page 86](#)).
- In SoMachine Basic / Twido, the messaging function is provided by the EXCHx instruction and the %MSGx function block. In the SoMachine 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.

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 SoMachine, 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 SoMachine, 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 SoMachine application, you have to declare this attribute keyword (*see page 539*) 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 SoMachine, there are 4 predefined watchlists (*see page 430*) (**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, SoMachine creates a trace object. You can view these variables in the trace editor (*see page 465*).

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 SoMachine project.

The following restrictions apply to the naming of symbols:

If...	Then ...
a symbol name does not comply with the naming rules of SoMachine,	the name of the symbol is modified.
a symbol name is equal to a keyword of SoMachine,	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 SoMachine 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 (<i>see SoMachine, Menu Commands, Online Help</i>).
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 Pertaining to 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 SoMachine. 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 SoMachine. In SoMachine Basic, the parameters of a function block can be accessed directly by the application program, for example, %HSC.P = 100. In SoMachine, 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.

Restriction	Solution
The behavior of counters differs in SoMachine 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 SoMachine: • 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 SoMachine: 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 Pertaining to the Conversion of PTO Function Blocks %PTO and %MC_XXXX

For M241:

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

Constraints Pertaining to 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 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 SoMachine 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 <code>GRAF CET_STATES</code>	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 <code>STATES</code> that contains 1 bit for each Grafcet state. Each bit represents the current value of the corresponding Grafcet state (%Xi object). 1 variable <code>ACTIVATE_STATES</code> that contains 1 bit for each Grafcet state. If the bit is TRUE, the Grafcet state is activated in the next cycle. 1 variable <code>DEACTIVATE_STATES</code> 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: <code>Init</code> initializes the Grafcet steps to their initial states. It is executed when the system bit %S21 is set by the application program. <code>Reset</code> 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 current state.
- A Grafcet deactivation instruction #Di is converted to setting the deactivation bit of state i and the deactivation bit of the current 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 SoMachine. For this reason, a graphical Grafcet POU is converted to an SFC program, as far 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. Carefully adapt the converted program.
Activation of multiple steps of an alternative branch is allowed.	Only one step of an alternative branch can be activated.	Carefully 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 all active steps have been executed.	Carefully 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 best practice to use TM3 modules. To convert the TM2 modules used in your Twido project into TM3 modules for the SoMachine 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. Carefully verify the converted module.
TM2AMI8HT	TM3AI8	–
TM2ARI8HT	–	The TM2 modules TM2ARI8HT, TM2ARI8LRJ, and TM2ARI8LT are 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		–

Source TM2 expansion module	Target TM3 expansion module	Further information
TM2AMM3HT	TM3TM3	–
TM2ALM3LT		It is possible that the behavior of the converted temperature module differs from the original module. Carefully 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. Carefully verify the converted module.
TM2AMM6HT	TM3AM6	–
TM2ARI8LRJ	–	The TM2 modules TM2ARI8HT, TM2ARI8LRJ, and TM2ARI8LT are not converted because there is no corresponding TM3 expansion module. You can replace this module by two TM3TI4 modules.
TM2ARI8LT	–	The TM2 modules TM2ARI8HT, TM2ARI8LRJ, and TM2ARI8LT are 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 SoMachine project, note their position in the tree structure: If TM3 nodes are located below TM2 nodes in the tree structure, this is reported as a detected **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 SoMachine, 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 (%IWN\$X)	Both SoMachine Basic and SoMachine provide status information for a slave device, but the status values are different. The status logic is partly converted.
Channel Status (%IWN\$X.y)	SoMachine 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.

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:

<code>READ_IMM_IN</code> and <code>WRITE_IMM_OUT</code> instructions (M221 controllers)	<code>GetImmediateFastInput</code> and <code>PhysicalWriteFastOutputs</code> functions (M241 controllers)
Access to all local inputs and outputs.	Access only to fast inputs and outputs.
<code>WRITE_IMM_OUT</code> writes a single bit (similar to the read function). <code>WRITE_IMM_OUT</code> returns an error code.	<code>PhysicalWriteFastOutputs</code> writes fast outputs at the same time. <code>PhysicalWriteFastOutputs</code> only returns the information on which outputs have actually been written.
The error codes of <code>READ_IMM_IN</code> and <code>GetImmediateFastInput</code> differ.	
<code>READ_IMM_IN</code> updates the input object (<code>%I0.x</code>).	<code>GetImmediateFastInput</code> 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 SoMachine application manually.

During conversion, one variable is created for each related I/O object in order to allow the SoMachine 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 *SoMachine, 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 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).

Chapter 6

Common Device Editor Dialogs

What Is in This Chapter?

This chapter contains the following sections:

Section	Topic	Page
6.1	Device Configuration	96
6.2	I/O Mapping	141

Section 6.1

Device Configuration

What Is in This Section?

This section contains the following topics:

Topic	Page
General Information About Device Editors	97
Controller Selection	99
Communication Settings	114
Configuration	118
Applications	119
Files	121
Log	123
PLC Settings	126
Users and Groups	128
Task Deployment	139
Status	140
Information	140

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 execute the **Edit Object** command via the context menu or via the **Project** menu.

The **Tools → Options → Device editor** dialog box allows you to make the generic device configuration views invisible (*see page 118*).

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
Controller Selection (<i>see page 99</i>)	Configuration of the connection between programming system and a programmable device (controller). This is the default tab for SoMachine V4.0 and later versions. SoMachine V3.1 and earlier versions use the Communication Settings tab by default.
Communication Settings (<i>see page 114</i>)	Configuration of the connection between programming system and a programmable device (controller). This is the default tab for SoMachine V3.1 and earlier versions. SoMachine V4.0 and later versions use the Controller Selection tab by default.
Configuration (<i>see page 118</i>)	Display or configuration of the device parameters.
Applications (<i>see page 119</i>)	List of applications currently running on the controller. Refer to the description in the <i>Program</i> chapter (<i>see page 151</i>).
Files (<i>see page 121</i>)	Configuration of a file transfer between host and controller.
Log (<i>see page 123</i>)	Display of the controller log file.
PLC settings (<i>see page 126</i>)	Configuration of: • application noticed for I/O handling • I/O behavior in stop status • bus cycle options
Users and Groups (<i>see page 892</i>)	User management concerning device access during runtime.
Access Rights (<i>see page 895</i>)	Configuration of the access rights on runtime objects and files for the particular user groups.

Tab	Description
Task Deployment (<i>see page 139</i>)	Display of inputs and outputs assigned to the defined task - used for troubleshooting.
Status (<i>see page 140</i>)	Device-specific status and diagnostic messages.
Information (<i>see page 140</i>)	General information on the device (for example: name, provider, version).
I/O Mapping (<i>see page 141</i>)	Mapping of the input and output channels of an I/O device on project (application) variables.

Controller Selection

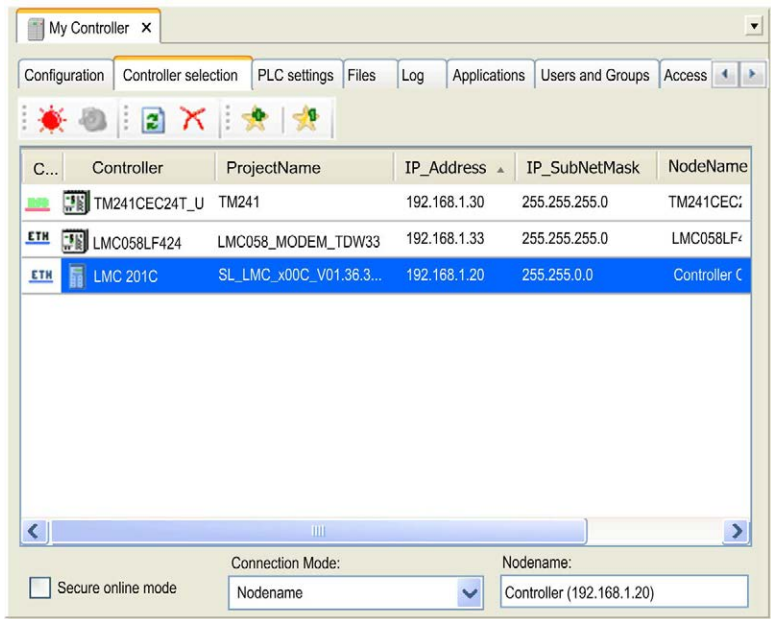
Overview

The **Controller selection** view of the device editor provides access to the Network Device Identification service. It allows you to scan the Ethernet network for available devices (such as controllers, HMI devices) and to display them in a list. In this view, 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 of SoMachine. 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* ([see page 807](#)).

The **Controller selection** view is only visible if the communication between controller and programming system is established by using the IP address. This is the default setting for SoMachine V4.0 and later versions. You can select between communication establishment via IP address and via active path in the **Project Settings** → **Communication settings** dialog box. If the option **Dial up via "IP-address"** is selected, the view **Controller selection** is displayed in the device editor. Otherwise, the **Communication settings** view is displayed.

Controller selection view of the device editor



The **Controller selection** view provides the following elements:

- buttons in the toolbar
- list providing information on the available controllers
- option, list, and text box at the bottom of the view

Description of the Buttons in the Toolbar

The following buttons are available in the toolbar:

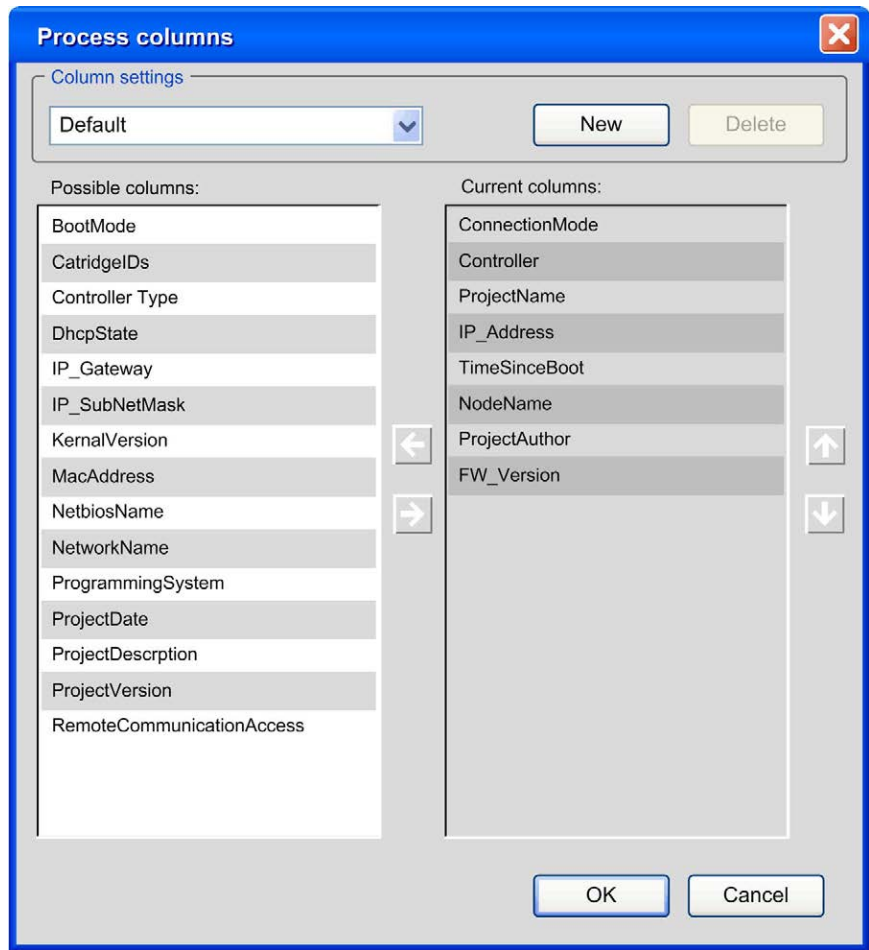
Button	Description
Optical	Click this button to cause the selected controller to indicate an optical signal: It flashes a control LED quickly. This can help you to identify the respective controller if many controllers are used. The function stops on a second click or automatically after about 30 seconds. NOTE: The optical signal is issued only by controllers that support this function.
Optical and acoustical	Click this button to cause the selected controller to indicate an optical and an acoustical signal: It starts to beep and flashes a control LED quickly. This can help you to identify the respective controller if many 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.
Update	Click this button to refresh the list of controllers. A request is sent to the controllers in the network. Any controller that responds to the request is listed with the current values. Pr-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. The Update button corresponds to the Refresh list command that is provided in the context menu if you right-click a controller in the list. To refresh the information of a selected controller, the context menu provides the command Refresh this controller. This command requests more detailed information from the selected controller. NOTE: The Refresh this controller command can also refresh the information of other controllers.

Button	Description
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 all controllers marked as inactive controllers simultaneously from the list. NOTE: A controller can be marked as inactive even if this is not the case. The context menu that opens if you right-click a controller in the list provides 2 other commands for removing controllers: • The Remove selected controller from list command allows you to remove only the selected controller from the list. • The Remove all controllers from list command allows you to remove all controllers simultaneously from the list.
New Favorite... and Favorite 0	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 0) to select or deselect it. If you have not selected a favorite, all detected controllers are visible. You can also access Favorites via the context menu. It opens upon 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 **Controller selection** view 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.

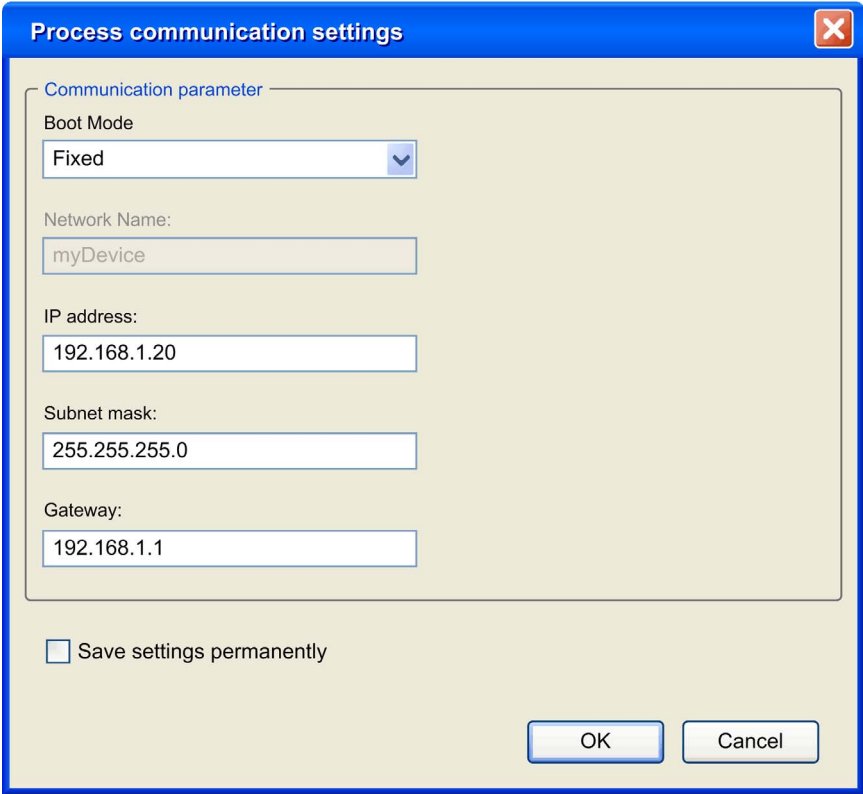
To achieve this, right-click the header of a column to open the **Process columns** dialog box.



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.

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 execute the command Process communication settings... from the context menu. Result: The Process communication settings... dialog box opens with the current 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 text box contains the IP address of the controller. It is a unique address that consists of 4 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 4 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 4 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

Secure online mode Option

The **Secure online mode** option causes SoMachine 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 secure online mode and thereby suppressing the display of this message, uncheck 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 XBT HMI 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.

Other devices, such as XBT HMI controllers, do not create a unique device name automatically. In this case, assign a unique device name (**NodeName**) as follows:

Step	Action
1	Right-click the controller in the list and execute the command Change device name... from the context menu. Result: The Change device name dialog box opens.
2	In the Change device name dialog box, enter a unique device name in the text box New .
3	Click the OK button to confirm. Result: The device name you entered is assigned to the controller and will be 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 **Controller selection** view allows you to select a format for the connection address you have to enter in the **Address** field.

The following formats are supported:

- Automatic (*see page 106*)
- Nodename (*see page 107*)
- IP Address (*see page 107*)
- Nodename via NAT (Remote TCP) (*see page 108*) (NAT = network address translation)
- IP Address via NAT (Remote TCP) (*see page 108*)
- Nodename via Gateway (*see page 109*)
- IP Address via Gateway (*see page 110*)
- Nodename via MODEM (*see page 112*)

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.

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** textbox if you switch to **Connection Mode → Automatic**.

Example:

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

Connection Mode:	NAT Address/Port	Target Nodename:
Nodename via NAT (Remote TCP) ▼	10.128.158.106 / 1105	MyTM241 (192.168.1.55)

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

Connection Mode:	Address:
Automatic ▼	enodename3://10.128.158.106:1105,MyTM241 (192.168.1.55)

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

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** text box.

NOTE: Do not use spaces at the beginning or end of the **Address**.
Do not use commas (,) in the **Address** text 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 text box is filled automatically if you double-click a controller in the list of controllers.

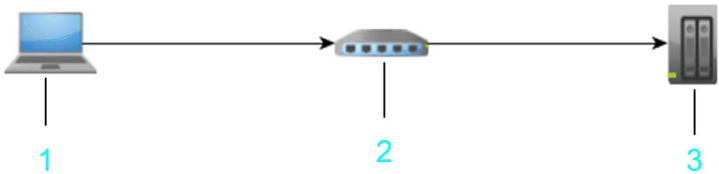
Example: **IP Address:** 190.201.100.100

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** text box.

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 / HMI
- 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.

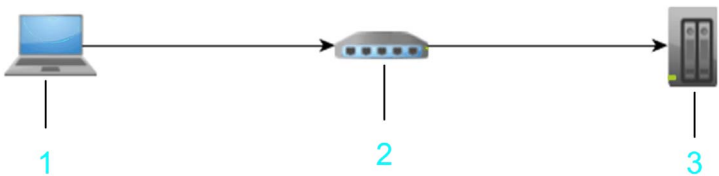
Do not use spaces at the beginning or end and do not use commas in the **Target Nodename** text box.

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 itself.

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 / HMI

- 2 NAT router
- 3 target device

Example: **NAT Address/Port:** 10.128.154.206/1217**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 SoMachine gateway port **1217** is used.

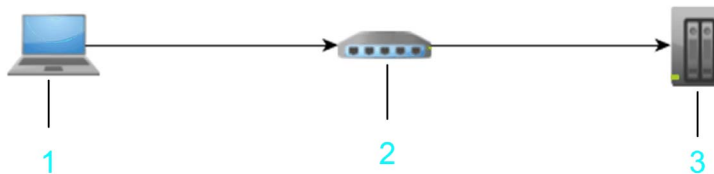
Enter a valid IP address (format <Number>.<Number>.<Number>.<Number>) for the **Target IP Address**.

The information you enter is interpreted as a URL 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)) or by calling a service on each scanned device of the 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.

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 a SoMachine gateway router in the network. Enter the nodename of the controller, and the IP address or host name and port of the SoMachine gateway router.



- 1 PC / HMI
- 2 PC / HMI / devices with installed SoMachine gateway
- 3 target device

Example: **Gateway Address/Port:** 10.128.156.28/1217**Target Nodename:** MyM238

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

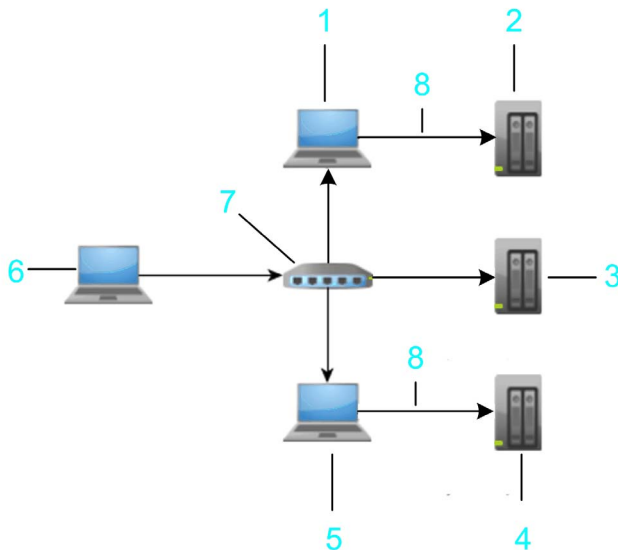
Enter the port of the gateway router to be used. Otherwise, the default SoMachine gateway port **1217** is used.

Do not use spaces at the beginning or end and do not use commas in the **Target Nodename** text 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 SoMachine gateway topology it is the root node itself or a child node of the root node.

NOTE: The SoMachine gateway can be located on the target controller, destination PC, or the local PC itself, making it possible to connect to a device that has no unique nodename but resides in a subnet behind a SoMachine network.

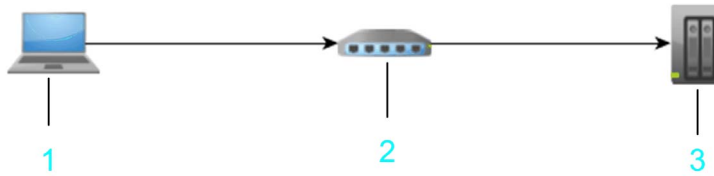
The graphic shows an example that allows a connection from the PCI / HMI to the target controller 3 (item 4 in the graphic) by using the address of hop PC2 (item 5 in the graphic) that must have a SoMachine gateway installed.



- 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

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 a SoMachine gateway router in the network. Enter the IP address of the controller, and the IP address or host name and port of the SoMachine gateway router.



- 1 PC / HMI
- 2 PC / HMI / devices with installed SoMachine 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:**.

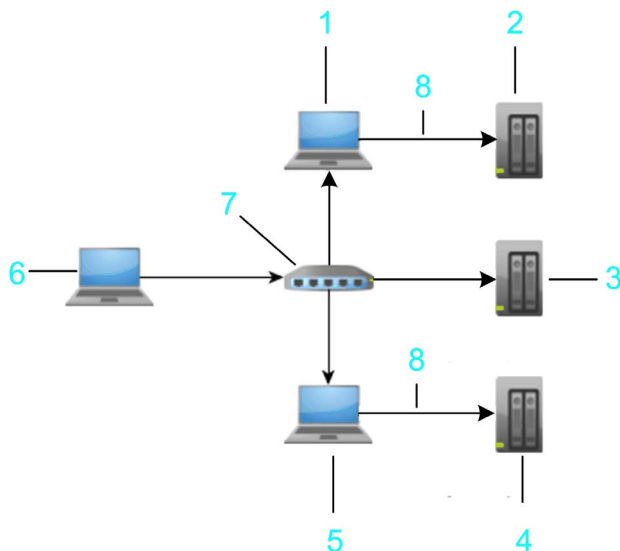
Enter the port of the gateway router to be used. Otherwise, the default SoMachine gateway port **1217** is used.

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)) or by calling a service on each scanned device of the gateway.

NOTE: The SoMachine gateway can be located on the target controller, destination PC, or the local PC itself. It is therefore possible to connect to a device that has no unique nodename but resides in a subnet behind a SoMachine network.

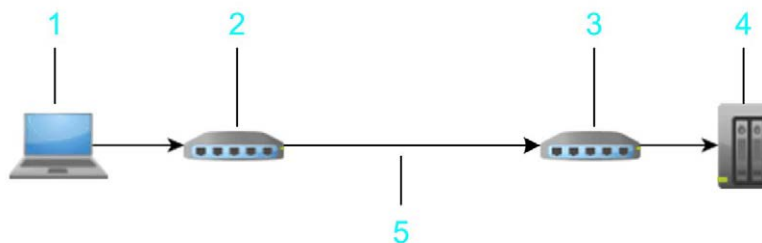
The graphic shows an example that allows a connection from hop PC2 (item 5 in the graphic) that must have a SoMachine gateway installed to the target controller 3 (item 4 in the graphic).



- 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 / HMI
- 7 router
- 8 Ethernet

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 / HMI

- 2 PC / HMI / 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 SoMachine gateway is stopped and restarted, any connection of the local gateway is terminated. SoMachine displays a message 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:** text box to connect to a specific controller.

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

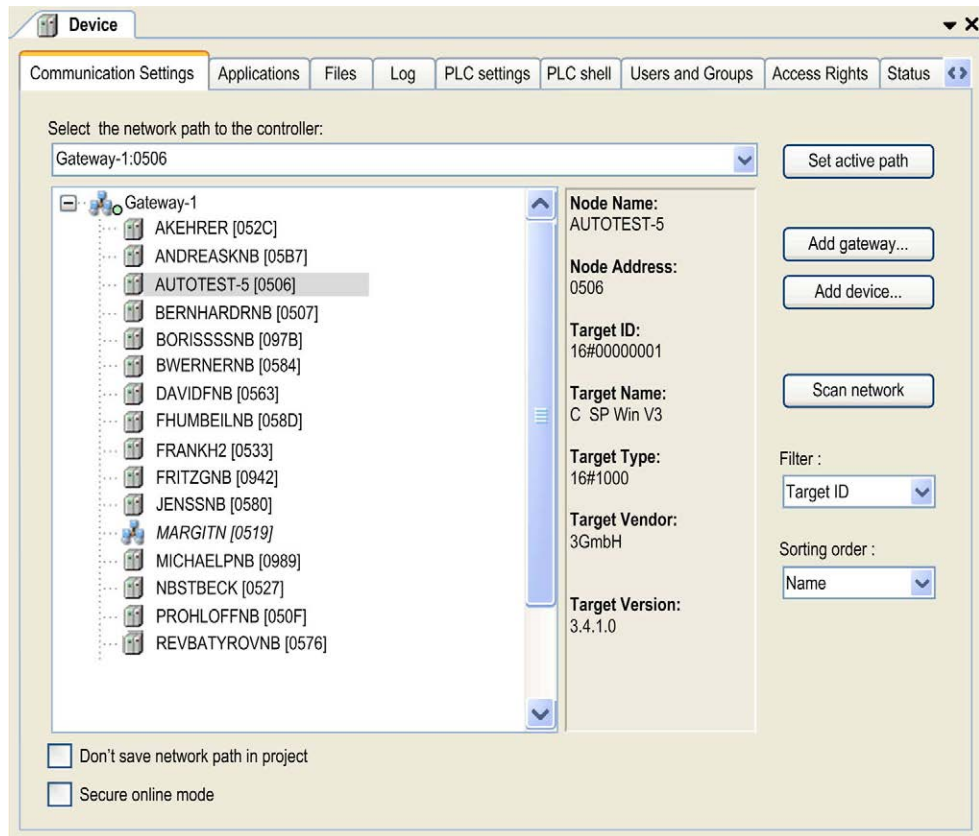
Communication Settings

Overview

The **Communication Settings** view of the device editor serves to configure the parameters for the communication between device and programming system.

It is only visible if the communication between device and programming system is established by using the active path. This is the default setting for SoMachine V3.1 and earlier versions. You can select between communication establishment via active path and IP address in the **Project Settings** → **Communication settings** dialog box. If the option **Dial up via "IP-address"** is selected, the view **Controller selection** is displayed in the device editor instead of the **Communication settings** view. This is the default setting for SoMachine V4.0 and later versions.

Communication Settings view of the device editor



This view is divided in 2 parts:

- The left part shows the currently 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 in the right part of the window:

Example:

Node Name: Gateway-1

Port: 1217

IP-Address: 127.0.0.1

Driver: TCP/IP

When the gateway is running, a green bullet is shown before the **Gateway** node; otherwise, a red bullet is displayed. The bullet is gray if the gateway has not been contacted yet (depending on some communication protocols, it is not allowed to poll the gateway, so the status cannot be displayed).

Indented below the **Gateway** node (open/close via the +/- sign), you will see entries for all 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 currently configured in the project, are displayed in gray font. To obtain an up-to-date list of the currently 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 **Target ID**, **Target Name**, **Target Type**, **Target Vendor**, and **Target Version** are shown.

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

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 view:

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

Description of the Buttons / Commands

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

Button / Command	Description
Set active path	This command sets the currently 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 current configuration. See the description of the Add Gateway command.
Add device...	This command opens the Add Device dialog box where you can manually define a device to be added to the currently selected gateway entry (Consider the Scan network functionality). See the description of the Add Device... command.
Edit Gateway...	This command opens the Gateway dialog box for editing the settings of the currently selected gateway. See the description of the Edit Gateway... command.
Delete selected Device	This command removes the selected device from the configuration tree. See the description of the Delete selected Device command.
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 complemented by their name. The scan refers to devices below that gateway in whose tree an entry is currently selected. See the description of the Scan for Device by Address command.
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 complemented by their unique node address. The scan refers to devices below that gateway in whose tree an entry is currently selected. See the description of the Scan for Device by Name command.

Button / Command	Description
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 complimented by their name. The scan refers to devices below that gateway in whose tree an entry is currently selected. See the description of the Scan for Device by IP Address command.
Connect to 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 <i>Gateway.cfg</i> . See the description of the Connect to local Gateway... command.
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. See the description of the Scan Network command.

Description of the Options

2 options are available below the tree structure:

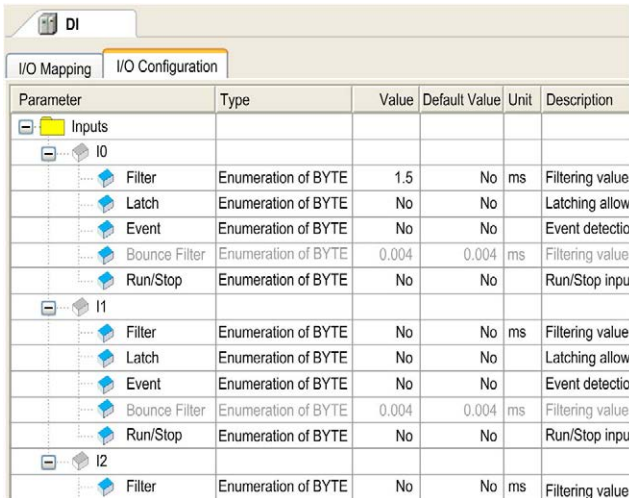
Option	Description
Don't save network path in project	Activate this option if the current network path definition should not be stored in the project, but in the local option settings on your computer. Therefore, the path setting is restored if the project is reopened on the same computer. It will have to be redefined if the project is used on another system.
Secure online mode	Activate this option if, for security reasons, 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



Parameter	Type	Value	Default Value	Unit	Description
Inputs					
I0					
Filter	Enumeration of BYTE	1.5	No	ms	Filtering value
Latch	Enumeration of BYTE	No	No		Latching allow
Event	Enumeration of BYTE	No	No		Event detectio
Bounce Filter	Enumeration of BYTE	0.004	0.004	ms	Filtering value
Run/Stop	Enumeration of BYTE	No	No		Run/Stop input
I1					
Filter	Enumeration of BYTE	No	No	ms	Filtering value
Latch	Enumeration of BYTE	No	No		Latching allow
Event	Enumeration of BYTE	No	No		Event detectio
Bounce Filter	Enumeration of BYTE	0.004	0.004	ms	Filtering value
Run/Stop	Enumeration of BYTE	No	No		Run/Stop input
I2					
Filter	Enumeration of BYTE	No	No	ms	Filtering value

The view contains the following elements:

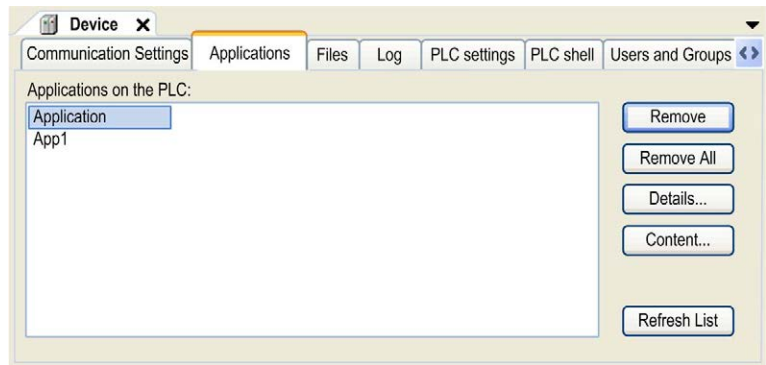
Element	Description
Parameter	parameter name, not editable
Type	data type of parameter, not editable
Value	Primarily, the default value of the parameter is displayed directly or by a symbolic name. If the parameter can be modified (this depends on the device description, non-editable parameters are displayed 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 standard 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

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.

Applications view of the device editor



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). If no scan has been executed yet or if a scan is not possible because no gateway is configured (<i>see page 114</i>) for a connection, an appropriate 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.

Element	Description
Content	If, in the View → Properties → Application build options, the option Download Application Info is activated for the application object (see <i>SoMachine</i>, <i>Menu Commands</i>, <i>Online Help</i>), 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 a more granular 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:

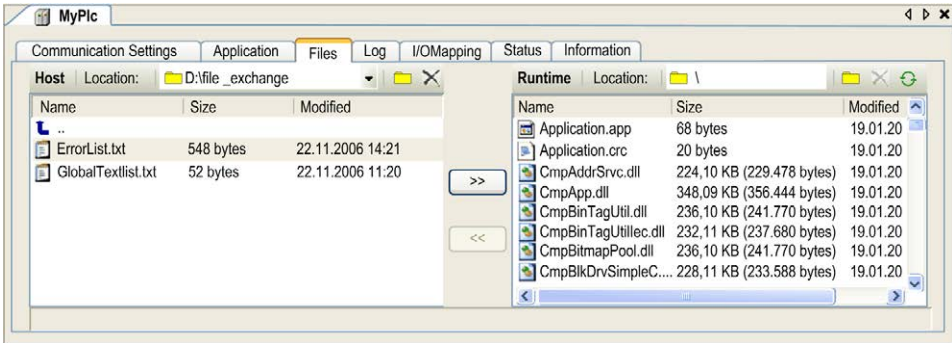
- The list of applications on the controller is compared with those available in the project. If inconsistencies are detected, the appropriate dialog boxes are displayed for either loading the applications not yet available on the controller, or for removing other applications from the controller.
- 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, an appropriate 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, an appropriate message (**signature mismatch(es)**) will be generated in a message box as well as in the **Messages** view if the option **Download** is selected.
- If in the **View → Properties → Application build options** the option **Download Application Info** is activated, additionally information on the content of the application will be loaded to the controller. See the description of the **Content** button in the previous table.

Files

Overview

The **Files** view of the device editor serves to transfer files between the host and the controller. This means 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.

Files view of the device editor



This view is divided in 2 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.
Location	Specifies the folder of the respective file system that will be used for the file transfer. Selects an entry from the list or browse in the file system tree.

Element	Description
<< >>	Select the files to be copied in the file system tree. You can select several files simultaneously or you can select a folder to copy all files it contains. To copy the selected files from the Host to the Runtime directory, click >>. To copy the selected files from the Runtime to the Host directory, click <<. If a file is not yet available in the target directory, it will be created there. If a file with the given name is already available and is not write-protected, it will be overwritten. If it is write-protected, an appropriate message will be generated.

Log

Overview

The **Log** view of the device editor serves 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

Log view of the device editor

The screenshot displays the 'Log' view within the 'Device' editor. The interface includes a top navigation bar with tabs: Communication Settings, Applications, Files, **Log**, PLC settings, PLC shell, Users and Groups, Access Rights, and Task deployment. Below the tabs, there are checkboxes for 'Offline-Logging' and 'UTC Time'. A status bar shows '2 warning(s)', '1 error(s)', '0 exception(s)', and '122 information(s)'. The main area is a table of log entries.

Severity	Time Stamp	Description	Component
Warning	20.10.2011 07:25:5...	VisuInfoTuple not found for RegisterClient, ExtId: 31909, Applicati...	CmpVisuHandler
Warning	20.10.2011 07:25:5...	VisuInfoTuple not found for RegisterClient, ExtId: 31907, Applicati...	CmpVisuHandler
Information	20.10.2011 05:59:5:0	Setting router 2 address to (014c:0001)	CmpRouter
Information	20.10.2011 05:59:1:0	Control ready	CM
Information	20.10.2011 05:59:1:0	Setting router 2 address to (0001)	CmpRouter
Information	20.10.2011 05:59:1:0	Setting router 1 address to (2ddc:c0a8:654c)	CmpRouter
Information	20.10.2011 05:59:1:0	Setting router 0 address to (054c)	CmpRouter
Information	20.10.2011 05:59:1:0	Local address (BlkDrvShm) set to 1	CmpBlkDrvShm
Information	20.10.2011 05:59:1:0	Network interface BlkDrvShm registered	CmpRouter
Information	20.10.2011 05:59:1:0	Network interface BlkDrvTcp registered	CmpRouter
Information	20.10.2011 05:59:1:0	Local network address: 192.168.101.76	CmpBlkDrvTcp
Information	20.10.2011 05:59:1:0	Network interface either 0 registered	CmpRouter
Information	20.10.2011 05:59:1:0	Network interface: 192.168.101.76 255.255.252.0	CmpBlkDrvUdp
Information	20.10.2011 05:59:1:0	Running as network client	CmpChannelMgr
Information	20.10.2011 05:59:1:0	Running as network server	CmpChannelMgr
Information	20.10.2011 05:59:1:0	*****	CmpWebServer
Information	20.10.2011 05:59:1:0	Root directory : .\visu	CmpWebServer
Information	20.10.2011 05:59:1:0	Port : 8080	CmpWebServer
Information	20.10.2011 05:59:1:0	IP-Address: 127.0.0.1	CmpWebServer
Information	20.10.2011 05:59:1:0	© Copyright by 3S – Smart Software Solutions GmbH, 2011	CmpWebServer
Information	20.10.2011 05:59:1:0	Web Server for 3S Runtime Systems	CmpWebServer
Information	20.10.2011 05:59:1:0	*****	CmpWebServer
Error	20.10.2011 05:59:1:0	WinPCap (www.winpcap.org) must be installed!	SysEthernet
Information	20.10.2011 05:59:1:0	Dynamic: CmpTargetVisuStub init, 0x00000053 3.5.0.0	CM

Description of the Elements

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

Element	Description
Severity	The events of the log are grouped in 4 categories: ● warning ● error ● exception ● information The buttons in the bar above the listing display the current 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, for example, 12.01.2007 09:48
Description	description of the event, for example Import function failed of <CmpFileTransfer>
Component	Here you can choose a particular component in order to obtain only displayed log entries regarding this component. The default setting is <All components> .
Logger	The selection list provides the available loggings. The default setting is <Default Logger> , which is defined by the runtime system.
	Updates the list.
	Exports the list to an XML file. The standard 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.
	Displays log entries stored in an XML file which may have been exported as described above. The standard 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 will be displayed in a separate window.
	Clears the current log table that is to remove all displayed entries.
Offline-Logging	This option is not used in SoMachine.

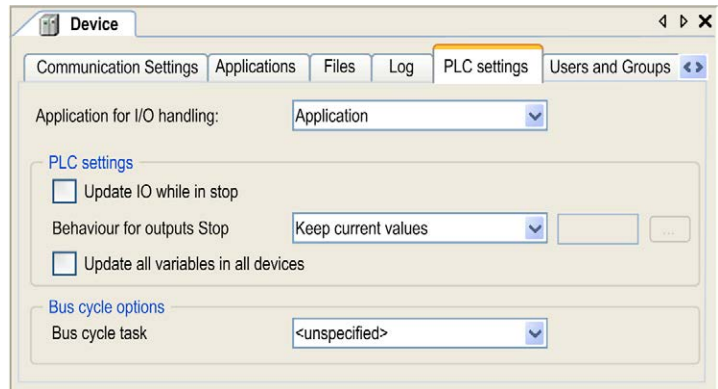
Element	Description
UTC Time	Activate this option to display the time stamp of the runtime system as it is (without conversion). If deactivated, the time stamp of the local time of the computer is displayed (according to the time zone of the operating system). NOTE: In order to display the time stamp 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).

PLC Settings

Overview

The **PLC settings** view of the device editor serves to configure some general settings for the controller.

PLC settings view of the device editor



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 SoMachine there is only one application available.
PLC settings area	
Update IO while in stop	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.

Element	Description
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 current 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.
Update all variables in all devices	If this option is activated, then for the devices of the current controller configuration, the I/O variables will get updated in each cycle of the bus cycle task. This corresponds to the option Always update variables. You can set it separately for each device in the I/O Mapping view (<i>see page 145</i>).
Bus cycle options area	
Bus cycle task	The selection list offers the tasks currently 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. This may be the task with the shortest cycle time, but as well it could be that with the longest cycle time.

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

 CAUTION
UNINTENDED EQUIPMENT OPERATION If you are not sure about the bus cycle task settings of the controller, do not set the Bus cycle task to <unspecified>, but select a dedicated task from the list. Failure to follow these instructions can result in injury or equipment damage.

Additional Setting

The setting **Generate force variables for IO mapping**: is only available if supported by the device. If the option is activated, for each I/O channel, which is assigned to a variable in the **I/O Mapping** dialog box, 2 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 (*see page 141*).

Users and Groups

Overview

The management of users and access-rights groups differs, depending on the controller you are using. For most devices supporting online user management (such as M258, M241, M251 and LMC••8 controllers), the **Users and Groups** view described in this chapter is used to manage user accounts and user access-rights groups and the associated access rights. This allows you to control the access on SoMachine projects and devices in online mode.

For managing user rights, you have to login as **Administrator** user.

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 measurements regularly and frequently.

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

NOTE: It is not intended that the **Users and Groups** feature be used to protect the SoMachine project against malicious access, but rather to help prevent mistakes from trusted users. If you want to protect your entire project, activate the option **Enable project file encryption** in the **Project Settings → Security** dialog box (*see SoMachine, Menu Commands, Online Help*). If you want to protect only a part of your code inside the project, put this code inside a compiled library (*see SoMachine, Menu Commands, Online Help*).

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

NOTE: You must establish user access-rights using SoMachine 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 *SoMachine 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.

For **Soft PLC** controllers, a specific **Users and Groups** view and a separate **Access Rights** view are displayed in the device editor. These specific views are described in the *User Management for Soft PLC* chapter (see page 889).

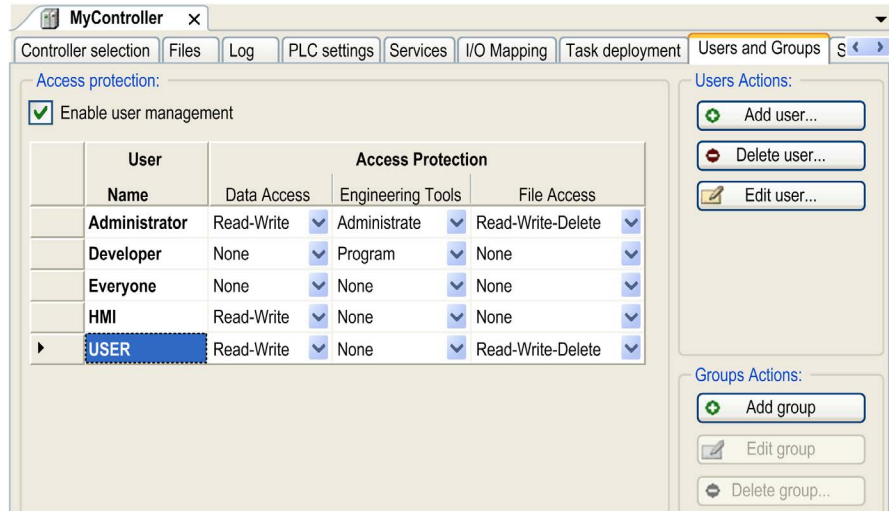
If you want that certain functions of a controller can only be executed by authorized users, the **Users and Groups** view allows you to define users, to assign access rights, and to require a user authentication at login.

To perform these actions, you can create users and configure their access rights to data, engineering tools, and files by using the buttons in the **User Actions** area. You can create user access-rights groups and configure each permission individually by using the buttons in the **Groups Actions** area.

Users and Groups View

The **Users and Groups** view allows you to manage the access of users on projects and devices. The users and the corresponding rights are valid for all devices of a project.

Users and Groups view of the device editor:



NOTE: When you apply the user rights configuration to a target, the new configuration may not be taken in account for connections that are already open. To help to make sure that the configuration is applied, close the connections to this target by either rebooting the target or disconnecting the cables (Ethernet and serial line) for at least 1 minute.

Access protection Area

The **Access protection** area of the **Users and Groups** view contains the **Enable user management** check box. This check box is by default disabled which means that user management is not active. Free access is provided to projects and devices. To be able to manage user accounts and user access-rights groups and to assign access rights, activate the **Enable user management** check box. To change user management settings, you must either login as **Administrator**, or have the **Administrate** user right for your user login. During first login as administrator, you are requested to change the default password.

Below the **Enable user management** check box, a list of the defined users (in the **User** column) and the access-rights groups they are granted (in the **Access Protection** columns sorted by type) is provided.

The list contains five users by default. You have limited access on these default users as listed in the table:

Default user	Description	Can be deleted	Name can be changed	Permissions can be changed	Default password (can be changed)
Administrator	User Administrator has no access restrictions; it is used for configuring user rights.	No	No	No	Administrator Must be changed with first login.
Everyone	User Everyone is used when you are not logged in. If you suppress permissions, you have to log in with a user which is granted permissions.	No	No	Yes	No password assigned.
USER	User USER is used by the web server to access the controller.	No	Yes	Yes	USER
HMI	User HMI is used by the HMI to connect to the controller.	No	Yes	Yes	HMI
Developer	To the user Developer there are granted suitable permissions for developing a project.	Yes	Yes	Yes	Developer

User access-rights are contained in groups, and are categorized in three areas, or access types, under **Access Protection**:

- **Data Access**: includes access to applications intending to read/write data from the device, such as HMI, OPC.
- **Engineering Tools**: includes access to programming tools, such as SoMachine and Controller Assistant.
- **File Access**: includes access to the internal or external file system, such as external media, SoMachine protocol, FTP, and Web.

Clicking the drop-down box of any defined user under any of the access type columns presents a list of the defined user access-rights groups that you can choose for the particular user and access type. For more information on user access-rights groups, refer to *Management of Access-Rights Groups* (see page 134).

Users Actions Area

The buttons are only available if the option **Enable user management** is activated.

Clicking a button opens a dialog box that prompts you to log in as user **Administrator** because this is the only user that is granted rights for user management.

The **User Actions** buttons are used to perform standard user management functions on users:

Button	Description
Add user...	Click this button to add a new user to the list. The dialog box Add user opens. Enter a User Name , a complete Name , a Description , and a Password . Repeat the password in the Confirm password field. To make the new user available for usage, enable the Activated option.
Delete user...	Click this button to delete the user selected in the list of the Access protection area.
Edit user...	Click this button to modify the user selected in the list of the Access protection area. The Edit user dialog box opens. It corresponds to the Add user dialog box (see above) containing the settings of the currently defined user. To make the user available for usage, enable the Activated option.

Change the default passwords for all the default users (**USER**, **HMI**, **Developer**). In addition, consider carefully the implications for giving any access-rights to the default user **Everyone**. On the one hand, granting access-rights to the user **Everyone** will allow you to run scripts from either the USB port or SD port (depending on your controller reference), while on the other hand it will allow anyone the access-rights you grant without first logging in.

WARNING

UNAUTHORIZED DATA ACCESS

- Immediately change any and all default passwords to new, secure passwords.
- Do not distribute passwords to unauthorized or otherwise unqualified personnel.
- Limit access-rights to the user **Everyone** to only those essential to your application needs.

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

NOTE: A secure password is one that has not been shared or distributed to any unauthorized personnel and does not contain any personal or otherwise obvious information. Further, a mix of upper and lower case letters and numbers offer greater security. You should choose a password length of at least seven characters.

Groups Actions Area

The buttons are only available if the option **Enable user management** is activated.

Clicking a button opens a dialog box that prompts you to log in as user **Administrator** because this is the only user that is granted rights for user management.

The buttons are used to perform access-rights management functions on groups:

Button	Description
Add group	Click this button to create a new group. The dialog box Custom groups opens.
Edit group	Click this button to modify a group. The Custom groups dialog box opens. From the Group to edit list, select the group you want to edit. The settings defined for the selected group are displayed in the tables below.
Delete group...	Click this button to delete the group selected in the list of the Access protection area. Note: Some default groups cannot be deleted (<i>see page 128</i>).

Access-Rights Groups for Access Protection Types

By default, a number of access-rights groups are predefined and are available from the drop-down lists under the individual **Access Protection** types for each defined user.

The table lists the available default access-rights groups per **Access Protection** type:

Access Protection type	Default Access-Rights Groups	Description
Data Access	None	Access is denied.
	Read	● Access to website of the device. ● Read unprotected data, such as: ○ Variables configured in OpcUaSymbolConfiguration (OPC UA client) ○ Variables configured in SymbolConfiguration (HMI, CoDeSys OPC Server) ○ Variables configured in WebDataConfiguration (web server) ○ Webvisualization
	Read-Write	● Access to website of the device. ● Read/write unprotected data, such as: ○ Variables configured in OpcUaSymbolConfiguration (OPC UA client) ○ Variables configured in SymbolConfiguration (HMI, CoDeSys OPC Server) ○ Variables configured in WebDataConfiguration (web server) ○ Webvisualization
	OpcUa	Access to OPC UA data (not HMI, nor web server). Assign this access right to the default user Everyone to allow anonymous access to data from the OPC UA server.
Engineering Tools	None	Login is denied.
	Monitor	● Login ● Upload source ● Read-only variables
	Operate	● Login ● Upload source ● Force / write variables
	Program	No restrictions, except user right configuration is not allowed.
	Administratrate	No restrictions, even user right configuration is allowed.
File Access	None	Access is denied.
	Read	Read unprotected files.
	Read-Write	Read/write unprotected files.
	Read-Write-Delete	Read/write/delete unprotected files.

Management of Access-Rights Groups

In addition to the default access-rights groups, you can create your own custom groups. You can configure access rights for specific tools or commands grouped under each **Access Protection** type (**Data Access**, **Engineering Tools**, **File Access**). To achieve this, click the **Add group** or **Edit group** button from the **Groups Actions** area. The **Custom groups** dialog box opens.

Actions	Permissions
Login	Default
Reset	Default
Run/Stop	Default
Logger	Deny
Variables	Default
Trace	Default
Project Settings	Default
PLC Settings	Deny
Download	Default
Online Change	Default
Image Download	Default
Upload Source	Default
Image Upload	Default
Update Device	Default
Project compilation	Default
Device Management	Default
Create Objects	Default
Developing tools	Default
Visualisation Tools	Default
Library management	Default
Recipe	Default

TargetName	Permissions
MyController	Default
MyController.Expert.PowerDistribution	Default
MyController.Expert.DM72F0	Default
MyController.Expert.DM72F1	Default
MyController.Expert	Default
MyController.TM5.TM5_Manager.Embedded Bus.Module_1	Default
MyController.TM5.TM5_Manager.Embedded Bus.Module_2	Default
MyController.TM5.TM5_Manager.Embedded Bus	Default
MyController.TM5.TM5_Manager	Default
MyController.TM5	Default
MyController.Ethernet	Default
MyController.Serial Line.SoMachine_Network_Manager	Default
MyController.Serial Line	Default
MyController.CAN0.CANopen_Performance	Default
MyController.CAN0	Default
MyController.PLC Logic.Application.GVL	Default
MyController.PLC Logic.Application.Task Configuration.MAST	Default
MyController.PLC Logic.Application.Task Configuration	Default
MyController.PLC Logic.Application.Library Manager	Default
MyController.PLC Logic.Application	Default
MyController.PLC Logic	Default

Parameter	Description
Name	Enter a Name for the group using a maximum of 32 characters. The characters a–z, A–Z, 0–9 and the underscore character are allowed.

Parameter	Description
Access Type	Select one of the available Access Types per group: • Data Access • Engineering Tools • File Access The Actions and Permissions available for the selected Access Type are listed below on the left-hand side.
Reset to	To copy the user access rights from one group to another, select the group from the Reset to list. This resets the Actions and Permissions to the values of the group you have selected.
Actions / Permissions list	This list shows the Actions and Permissions available for the selected Access Type. Refer to the lists of actions and permissions per access type below.
TargetName / Permissions list	The TargetNames in this list are exactly those you find in the Devices tree of your project for the device you have selected. You can configure permissions for each target device individually. To achieve this, select the device in the list and select the suitable option (Default, Deny, Read only, Modifiable, Full) from the Permissions list. The Default value corresponds to the permissions of user Everyone.

The table lists the available **Actions** and **Permissions** you can assign for the **Access Type → Data Access**:

Available Actions	Permissions
OPC Server Allows the user to connect to a device within the OPC server by using login / password parameters.	• Deny • Read • Read - Write
HMI Allows the user to access Vijeo-Designer by using login / password parameters.	• Deny • Read • Read - Write
Web Server Allows the user to access the web server by using login / password parameters.	• Deny The user cannot access the web server. • Read The user can access the web server and read variables. • Read - Write The user can access the web server and read and write variables.

The table lists the available **Actions** and **Permissions** you can assign for the **Access Type** → **Engineering Tools**:

Available Actions	Permissions
Login Allows the user to connect to the device and to access the application. This permission is a prerequisite for any other online permission to be granted.	● Default ● Deny ● Grant
Reset	● Default Same permissions as user Everyone. ● Deny User cannot execute a reset of the device. ● Cold User can execute a cold reset of the device. ● Cold - Warm User can execute a cold and a warm reset of the device.
Run / Stop	● Default Same permissions as user Everyone. ● Deny User cannot execute a run / stop of the application. ● Grant User can execute a run / stop of the application.
Logger Allows the user to access the Log view of the device editor.	● Default Same permissions as user Everyone. ● Deny ● Grant
Variables	● Default Same permissions as user Everyone. ● Deny User cannot read/write any variables. ● Read User can read the variables of the application. ● Read - Write User can read, write, and force unprotected variables.
Trace Allows the user to access the trace device.	● Default Same permissions as user Everyone. ● Deny ● Grant

Available Actions	Permissions
Project Settings Allows the user to modify (read and write) the project information and parameters.	• Default Same permissions as user Everyone. • Deny • Grant
PLC Settings Allows the user to access the PLC Settings view of the device editor.	• Default Same permissions as user Everyone. • Deny • Grant
Download Allows the user to download all applications or the currently active application to the device.	• Default Same permissions as user Everyone. • Deny • Grant
Online Change Allows the user to execute the Online Change command.	• Default Same permissions as user Everyone. • Deny • Grant
Image Download Allows the user to download images.	• Default Same permissions as user Everyone. • Deny • Grant
Upload Source Allows the user to execute the Source Upload command.	• Default Same permissions as user Everyone. • Deny • Change
Image Upload Allows the user to upload images.	• Default Same permissions as user Everyone. • Deny • Change
Update Device Allows the user to execute the Update Device command.	• Default • Deny • Change
Project Compilation Allows the user to execute the Build and Build all command.	• Default Same permissions as user Everyone. • Deny • Change
Device Management Allows the user to add, edit, delete, update, refresh a device.	• Default Same permissions as user Everyone. • Deny • Change

Available Actions	Permissions
Create Object Allows the user to create objects in a project.	• Default Same permissions as user Everyone. • Deny • Change
Developing Tools Allows the user to modify applications, POUs, devices.	• Default Same permissions as user Everyone. • Deny • Change
Visualization Tools Allows the user to add and edit a web visualization and to create visualizations in the current project.	• Default Same permissions as user Everyone. • Deny • Change
Library Management Allows the user to perform library management, except accessing the list of libraries in a project and showing their properties.	• Default Same permissions as user Everyone. • Deny • Change
Recipe Allows the user to create, edit, send, upload, play a recipe and to load data from the current program to a recipe.	• Default Same permissions as user Everyone. • Deny • Change
Debugging Tools Allows the user to execute debugging commands, including breakpoints.	• Default Same permissions as user Everyone. • Deny • Change
SoftMotion Tools Allows the user to execute CAM and CNC commands.	• Default Same permissions as user Everyone. • Deny • Change
DTM Management Allows the user to execute field device tool (FDT) commands.	• Default Same permissions as user Everyone. • Deny • Change

The table lists the available **Actions** and **Permissions** you can assign for the **Access Type → File Access**:

Available Actions	Permissions
File system FTP Removable media Allows the user to access the file using FTP, SoMachine, or web.	• Deny • Read • Read - Write • Read - Write - Delete

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

Files Log PLC settings PLC shell Users and Groups Access Rights Licenses Task deployment Status < >			
I/O deployment for tasks:			
I/O channels	Main Task (0)	Bus Task (1)	
BK5120			
usiBK5120Out AT %QB0	☒		☒
usiBK5120In AT %IB0	☐		☒
Generic_XN_16DO			
usiGenericOut1 AT %QB1	☒		☐
usiGenericOut2 AT %QB2	☒		☐
Generic_XN_16DI			
usiGenericIn1 AT %IB1	☐		☒
usiGenericIn2 AT %IB2	☐		☒
Osicoder			
udiOsicoderIn AT %ID1	☐		☒
ILB_CO_24_DI16_DO16			
PhoenixOut1 AT %QB3	☒		☐
%QB4	☐		☒
%IB8	☐		☒
 = Bus cycle task			

The table shows the tasks sorted by their task priority. Click the column heading (task name) 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. In general, it is ill-advised to write output references in more than one task, as it makes the program difficult to debug and often 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 respective device; also on the used card and the internal bus system.

Information

Overview

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

Section 6.2

I/O Mapping

What Is in This Section?

This section contains the following topics:

Topic	Page
I/O Mapping	142
Working with the I/O Mapping Dialog	145
I/O Mapping in Online Mode	148
Implicit Variables for Forcing I/Os	149

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. This means that 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 ([see page 126](#)).

NOTE: If supported by the device, you can use the online configuration mode to access the I/Os of the hardware without having an application loaded beforehand. For further information, refer to the description of the Online Config Mode ([see SoMachine, Menu Commands, Online Help](#)).

See the following chapters:

- Working with the I/O Mapping Dialog ([see page 145](#))
- I/O Mapping in Online Mode ([see page 148](#))
- Implicit Variables for Forcing I/Os ([see page 149](#))

General Information on Mapping I/Os on Variables

Whether an I/O mapping can be configured for the current 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* ([see page 147](#)).

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 1 input.
- Instead of using the **I/O Mapping** view, you can also assign an address to a variable via the AT declaration ([see page 530](#)).

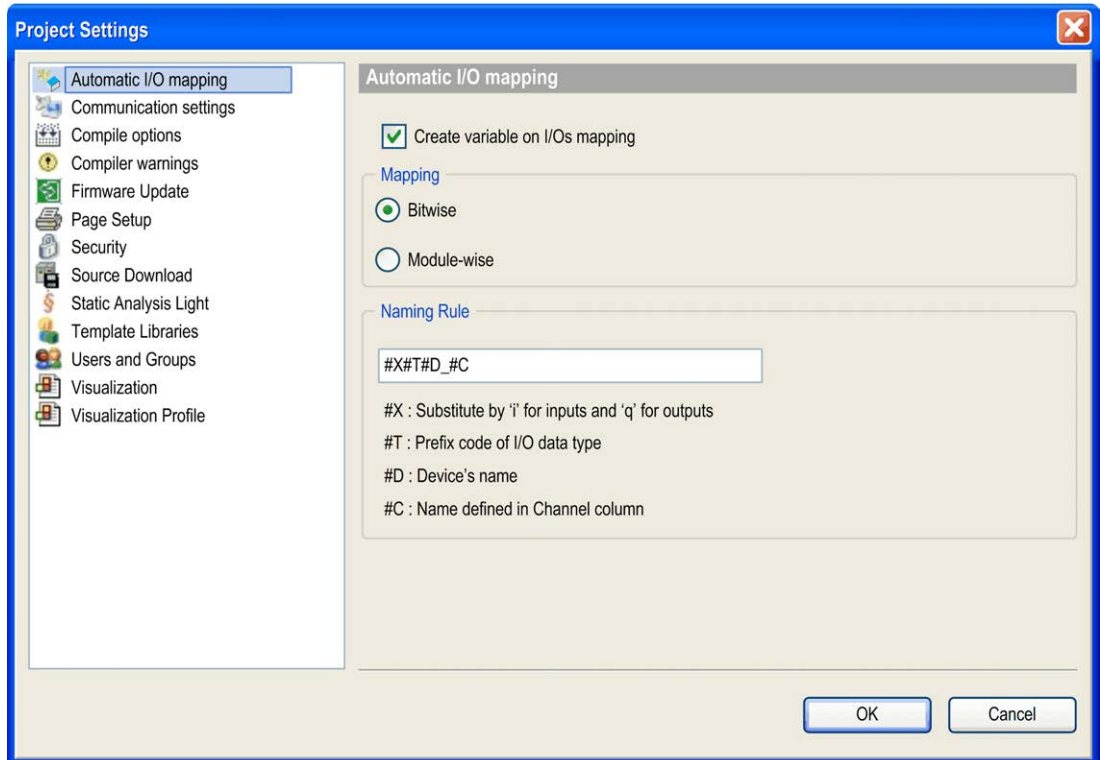
However, consider the following:

- You can use AT declarations only with local or global variables; not however, with input and output variables of POU's.
- The possibility of generating force variables for I/Os (refer to Implicit Variables for Forcing I/Os ([see page 149](#))) 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 SoMachine, Menu Commands, Online Help](#)). You can use them for forcing the input or output value during the commissioning of a machine, for example, via a visualization (HMI). Refer to the chapter *Implicit Variables for Forcing I/Os* ([see page 149](#)).

Automatic I/O Mapping

SoMachine V4.0 and later versions provide an automatic I/O mapping function. It 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.



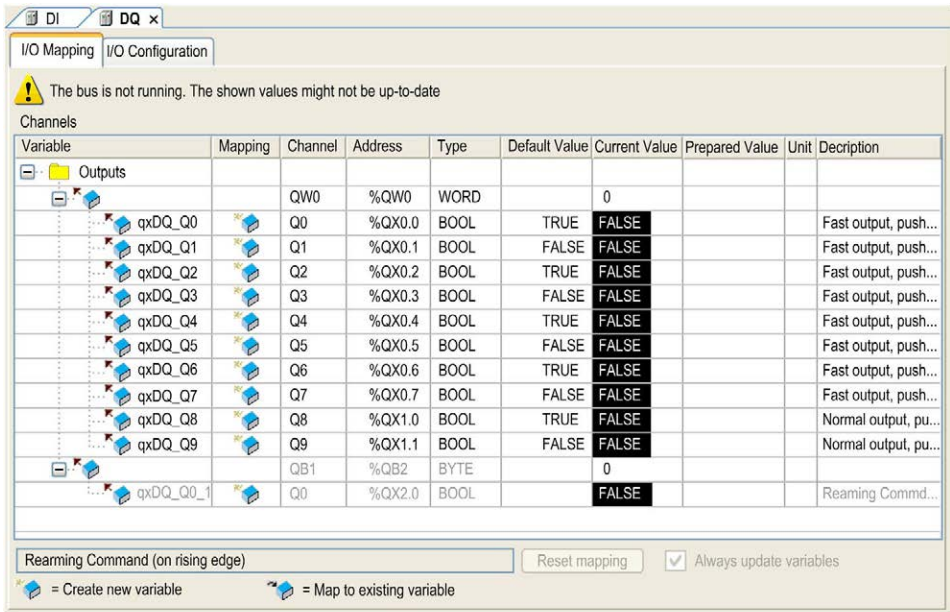
The dialog box provides the following elements:

Element	Description
Create variable on I/Os mapping	Select or deselect this option to activate or deactivate the automatic I/O mapping function.
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 respective data type of the variable in the variable name. The prefixes that are used for the different data types are listed in the <i>Recommendations on the Naming of Identifiers</i> chapter (see page 522). • 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

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 standard. It is indicated as IEC type in the device description. Otherwise, the entry of the table will be empty.
Unit	Unit of the parameter value, for example: ms for milliseconds
Description	Short description of the parameter
Current Value	Present value of the parameter, displayed in online mode

NOTE: Inputs and outputs that are not used in the controller code are not read by the controller in online mode. To indicate that the values displayed for these inputs or outputs may not be correct, they are marked with a gray background.

Configuration of the I/O Mapping

Perform the I/O mapping by assigning the appropriate 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, 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 entry with a subtree of bit channel entries in the mapping table (as shown in the figure below), 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:

DP parameters					
DP-Module Configuration					
DP-Module I/O Mapping					
Status					
Information					
Channels					
Variable	Mapping	Channel	Address	Type	Current V
		Output0	%QB2		
		Bit0	%QX2.0	BOOL	
 Profibus.PLC_PRG.xBool_4		Bit1	%QX2.1	BOOL	

- 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 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, it is recommended to do the mappings within the device configuration dialog.

NOTE: If a UNION is represented by I/O channels in the mapping dialog, 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, 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 sub-elements 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 sub-elements.

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	If this option is activated, the variables will be updated in each cycle of the bus cycle task (see page 126), no matter whether they are used or whether they are mapped to an input or output.

IEC Objects

This part of the tab is only available if implicitly an instance of the device object has been created, which can be accessed by the application (for example, in order to restart a bus or to poll information). Whether such an instance is available and how it can be used, depends on the controller and is described in its programming guide.

Bus Cycle Options

This configuration option will be available for devices with cyclic calls before and after reading inputs or outputs. It allows you to set a device-specific bus cycle task (*see page 126*).

Per default, the parent bus cycle setting will be valid (**Use parent bus cycle setting**). This means that is 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.

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), 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 can be necessary to force I/Os, for example, by an HMI visualization. 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, 2 variables will be generated according to the following syntax. Any empty spaces in the channel name will be 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 will be available in the input assistant in category **Variables** → **IoConfig_Globals_Force_Variables**. They can be used in any 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 define 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 (*see page 145*), then at a build (F11) of the application, the following variables will be generated and be 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) 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). 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.

Part III

Program

What Is in This Part?

This part contains the following chapters:

Chapter	Chapter Name	Page
7	Program Components	153
8	Task Configuration	227
9	Managing Applications	231

Chapter 7

Program Components

What Is in This Chapter?

This chapter contains the following sections:

Section	Topic	Page
7.1	Program Organization Unit (POU)	154
7.2	Function Block	181
7.3	Application Objects	200
7.4	Application	225

Section 7.1

Program Organization Unit (POU)

What Is in This Section?

This section contains the following topics:

Topic	Page
POU	155
Adding and Calling POUs	156
Program	160
Function	162
Method	165
Property	168
Interface	170
Interface Property	174
Action	177
External Function, Function Block, Method	179
POUs for Implicit Checks	180

POU

Overview

The term Program Organizational Unit (POU) is used for all programming objects (programs, function blocks, functions, etc.) which are used to create a controller application.

POU Management

POUs which are managed in the **Global** node of the **Applications tree** are not device-specific but they can be instantiated for the use on a device (application). For this purpose, program POUs must be called by a task of the respective application.

POUs which are inserted in the **Applications tree** explicitly below an application, can only be instantiated by applications indented below this application (child application). For further information, see the descriptions of the **Devices tree** ([see page 39](#)) and of the **Application** object ([see page 225](#)).

But POU also is the name of a certain sub-category of these objects in the **Add Object** menu. At this place, 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, 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, refer to the section *Adding POUs to an Application* ([see page 157](#)). The POUs you have created are added to the **Assets** view of the **Software catalog**.

You can add a POU available in the **Assets** view to the project in 2 different ways:

- Select a POU in the **Assets** view and drag it to the suitable node in the **Applications tree**.
- Select a POU in the **Assets** view and drag it to the logic editor view ([see page 255](#)).

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 POUs

Introduction

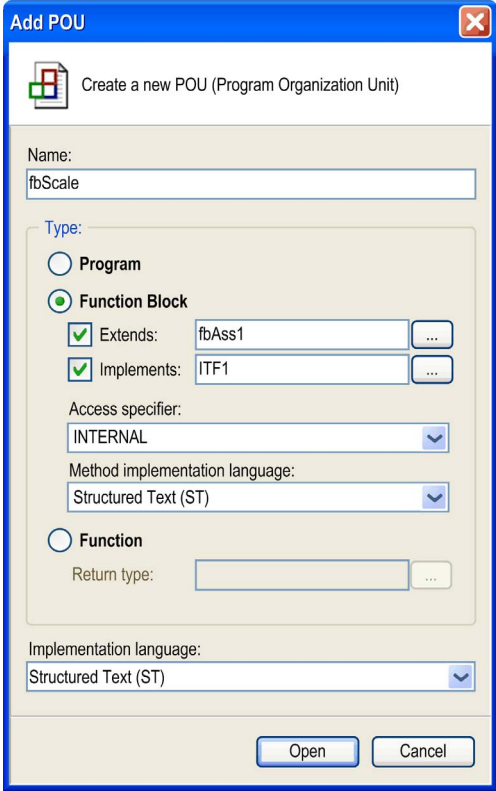
You can add Program Organization Units (POUs) to your application in the **Software catalog** → **Assets** or in the **Applications tree**.

The different types of POU 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.
- **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 to an Application

To add a POU to the application of the controller, proceed as follows:

Step	Action
1	In the Software catalog → Assets → POUs section, select an Application node, click the green plus button, and execute the command POU.... As an alternative, you can right-click the Application node of the controller and choose Add Object → POU. The 2 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 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 you want: ● Program ● Function Block: a. Select Extends and click the browser to select the block function you want in the Input Assistant. b. Click the OK button. c. Select Implements and click the browser to select the interface you want in the Input Assistant. d. Click the OK button. e. In the list box Method implementation language, select the programming language you want for editing the function block. ● Function: a. Click the browse button to select the Return type you want in the Input Assistant. b. Click the OK button.
4	From the list box Implementation Language , select the programming language you want for editing your program.
5	Click the Open button.

Already defined POUs are listed in the **Software catalog** → **Assets** → **POUs** section. You can add them to your application, by dragging them to the **Applications tree** and dropping them on an **Application** node. You can also drop a POU on the logic editor view.

Assigning POUs to a Task

At least 1 POU has to be assigned to a task. To add a POU 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 POU. In the Configuration tab, click Add Call. Alternatively, in the Applications tree select a task node you want to declare your POU and click the green plus button. Execute the command POU... 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 (Project) .
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 POU

POUs can call other POU. 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 ([see page 727](#)) 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* ([see page 180](#)).

Program

Overview

A program is a POU which returns one or several values during operation. All values are retained from the last time the program was run until the next.

Adding a Program

To assign 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 context 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 will open and you can start editing the program.

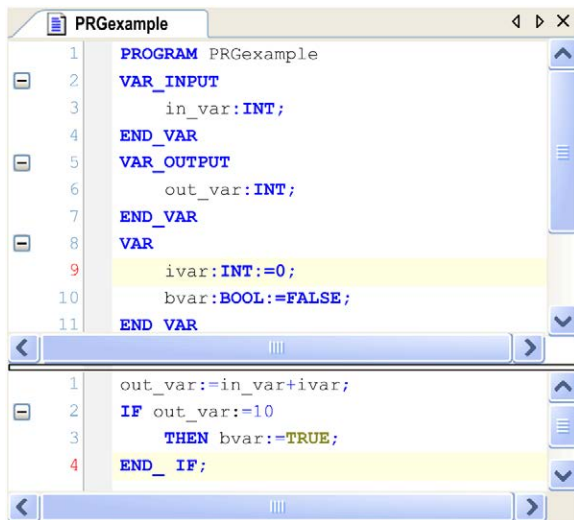
Declaring a Program

Syntax:

PROGRAM <program name>

This is followed by the variable declarations of input ([see page 536](#)), output ([see page 536](#)), 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 (*see page 162*) 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 changed, these changes will be retained until the program gets called again. This applies even if it will be 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 changed. The changes only are 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 (*see page 527*) 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 context 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>

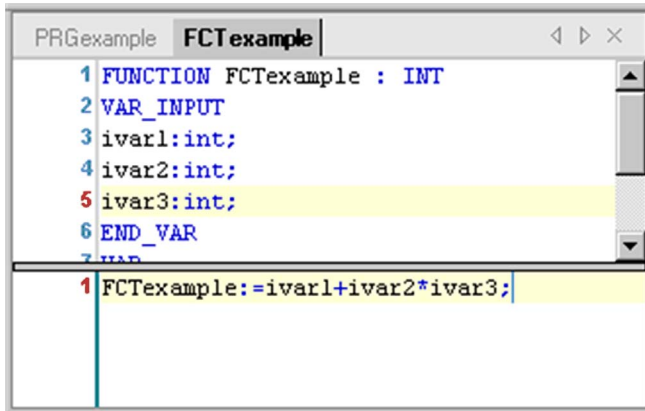
This is followed by the variable declarations of input and function variables.

Assign a result to a function. This means that 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.

The compiler generates appropriate messages if local variables declared as **RETAIN** or **PERSISTENT** are detected.

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;

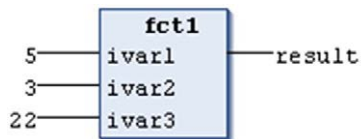
LD	5	
Fct	3	,
	22	
ST	result	

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 f
following:
fun(formal1 := actual1, formal2 := actual2);
  
```

According to the IEC 61131-3 standard, functions can have additional outputs. They must 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`. The return value of `fun` is written to the locally declared output variables (*see page 536*) (`VAR_OUTPUT`) `loc1` and `loc2`.

```

fun(in1 := 1, in2 := 2, out1 => loc1, out2 => loc2);
  
```


Method

Overview

The **Method** functionality is only available if selected in the currently used feature set (**Options → Features → Predefined feature sets**).

You can use methods to describe a sequence of instructions because they support object-oriented programming. Unlike a function, a method is not an independent POU, but must be assigned to a function block (*see page 181*). It can be regarded as a function which contains an instance of the respective function block. Such as a function it has a return value, an own declaration part for temporary variables and parameters.

Also as a means of object-oriented programming, you can use interfaces (*see page 170*) to organize the methods available in a project. An interface is a collection of method-prototypes. This means a method assigned to an interface only contains a declaration part, but no implementation. Further on in the declaration, no local and static variables are allowed, but only input, output and input/output variables. The implementation of the method is to be done in the function block which implements the interface (*see page 191*) and uses the method.

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 appropriate 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 context menu.

In the **Add Method** dialog box, enter a **Name**, the desired **Return Type**, the **Implementation Language**, and the **Access Specifier** (see below). 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 current namespace (the library).
- **FINAL:** No overwriting access on the method is possible. 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 older versions. For further information, refer to the SoMachine/CoDeSys compiler version mapping table in the SoMachine Compatibility and Migration User Guide (*see SoMachine Compatibility and Migration, User Guide*).

Click **Open** to confirm. The method editor view opens.

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 ([see page 170](#)).

Calling a Method

Method calls are also named virtual function calls. For further information, refer to the chapter *Method Invocation* ([see page 193](#)).

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). This means that the variables and function blocks declared in a method are reinitialized at each call of the method.
- In the body of a method, access to the function block instance variables is allowed.
- If necessary, use the THIS pointer ([see page 197](#)) which always points on the current instance.
- VAR_IN_OUT or VAR_TEMP variables of the function block cannot be accessed in a method.
- Methods defined in an interface ([see page 170](#)) 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* ([see page 193](#)).

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 (see page 546).
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 (see page 546).
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 (see page 549).

Properties ([see page 168](#)) and interface properties ([see page 174](#)) 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

The **Property** functionality is only available if selected in the currently used feature set (**Options → Features → Predefined feature sets**).

A property in extension to the IEC 61131-3 is available as a means of object-oriented programming. It consists of a pair of accessor methods (Get, Set). They are called automatically at a read or write access on the function block, which has got the property.

To insert a property as an object below a program (*see page 160*) or a function block (*see page 181*) 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 context menu.

In the **Add Property** dialog box specify the **Name**, **Return Type**, desired **Implementation Language**, and optionally an **Access Specifier**.

The same access specifiers are available as for methods (*see page 165*):

- **PUBLIC**
- **PRIVATE**
- **PROTECTED**
- **INTERNAL**
- **FINAL**

NOTE: Properties can also be used within interfaces.

Get and Set Accessors of a Property

2 special methods (*see page 165*), 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 POU** dialog box, when explicitly adding the accessor.

- The Set accessor is called when the property is written that is the name of the property is used as input.
- The Get accessor is called when the property is read that is the name of the property is used as output.

Example:

Function block FB1 uses a local variable `milli`. This variable is determined by the properties Get and Set:

Get example

```
seconds := milli / 1000;
```

Set example

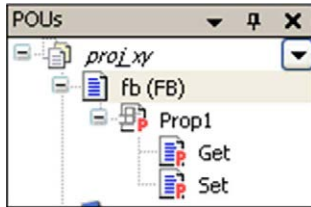
```
milli := seconds * 1000;
```

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 Prop1 is assigned to function block fb:



A property can have additional local variables but no additional inputs and - in contrast to a function ([see page 162](#)) or method ([see page 165](#)) - 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 ([see page 362](#)) or with help of a watch list ([see page 430](#)). The precondition for monitoring a property is the addition of the pragma {attribute 'monitoring:=variable'} (refer to the chapter Attribute Monitoring ([see page 578](#))) on top of its definition.

Interface

Overview

The **Interface** functionality is only available if selected in the currently used feature set (**Options → Features → Predefined feature sets**).

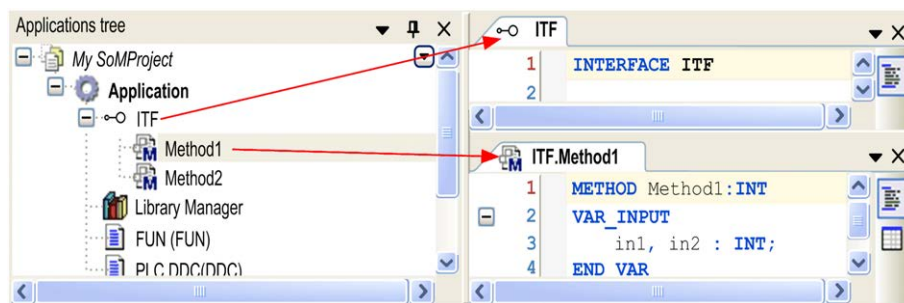
The use of interfaces is a means of object-oriented programming. An interface POU describes a set of methods (*see page 165*) and property (*see page 168*) prototypes. Prototype means that just declarations but no implementation is contained. An interface can be described as an empty shell of a function block (*see page 181*). It must be implemented (*see page 191*) in the declaration of the function block in order to get realized in the function block instances. Not until being part of a function block definition, it can be filled with the function block-specific programming code. 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

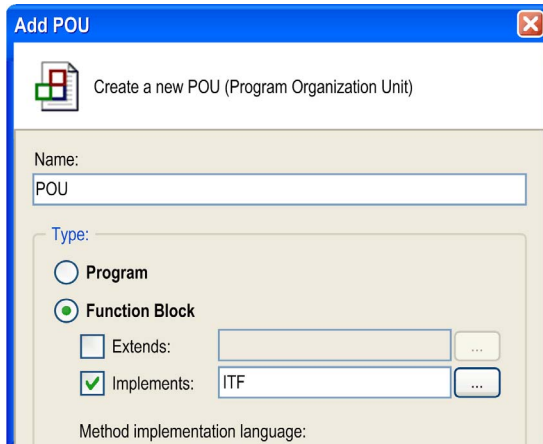
An interface IFT is inserted below an application. It contains 2 methods Method1 and Method2. Neither the interface nor the methods contain any implementation code. Just the declaration part of the methods is to be filled with the desired variable declarations:

Interface with 2 methods:



1 or multiple function blocks can now be inserted, implementing the above defined interface IFT.

Creating a function block implementing an interface



Add POU

Create a new POU (Program Organization Unit)

Name:
POU

Type:

☐ Program

☒ Function Block

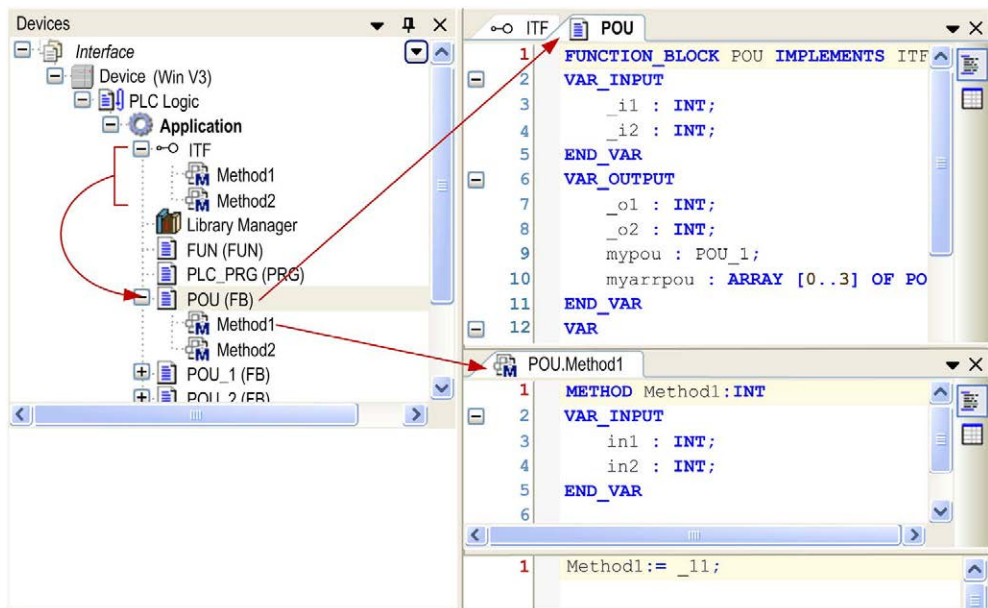
☐ Extends:

☒ Implements: ITF

Method implementation language:

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



The screenshot shows the IDE interface with the **Applications tree** on the left and the **POU** definition on the right.

Applications tree:

- Interface
 - Device (Win V3)
 - PLC Logic
 - Application
 - ITF
 - Method1
 - Method2
 - Library Manager
 - FUN (FUN)
 - PLC_PRG (PRG)
 - POU (FB)
 - Method1
 - Method2
 - POU_1 (FB)
 - POU_2 (FB)

POU Definition:

```

1 FUNCTION_BLOCK POU IMPLEMENTS ITF
2 VAR_INPUT
3   _i1 : INT;
4   _i2 : INT;
5 END_VAR
6 VAR_OUTPUT
7   _o1 : INT;
8   _o2 : INT;
9 mypou : POU_1;
10 myarrpou : ARRAY [0..3] OF PO
11 END_VAR
12 VAR

```

POU.Method1:

```

1 METHOD Method1:INT
2 VAR_INPUT
3   in1 : INT;
4   in2 : INT;
5 END_VAR
6

```

POU.Method1 Implementation:

```

1 Method1:= _i1;

```

An interface can extend other interfaces by using EXTENDS (see following example *Example for Extending an Interface* (see page 172)) in the interface definition. This is also possible for function blocks.

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 chapters *Property* (see page 168) and *Interface Property* (see page 174). 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

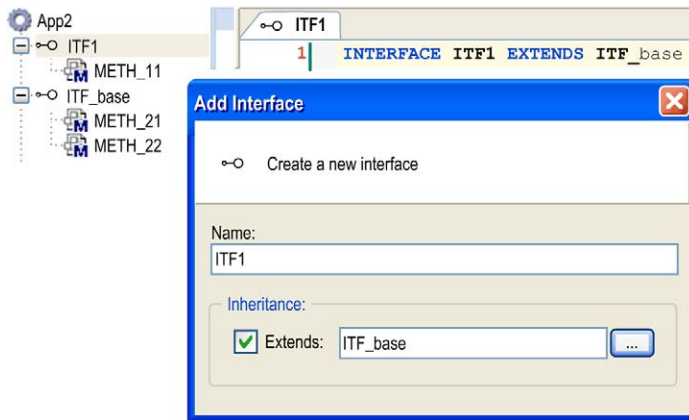
To add an interface to an application, select the **Application** node in the **Applications tree** or in the **Software Catalog → Assets**, click the green plus button and select **Add Other Objects... → Interface**. Alternatively, execute the command **Add Object → Interface**. If you select the node **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 (see page 188) 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** or in the **Software Catalog** → **Assets** 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).

Interface Property

Overview

A property, available as a means of object-oriented programming, can - besides with methods and programs - also be used within the definition of an interface (*see page 170*). In this case, it is named interface property. To add it to the interface selected in the **Applications tree**, click the green plus button, and execute the command **Interface Property....** Alternatively, right-click the interface node, and execute the command **Add Object → Interface property** from the context menu.

For further information on a property and its methods, refer to *Property (see page 168)*

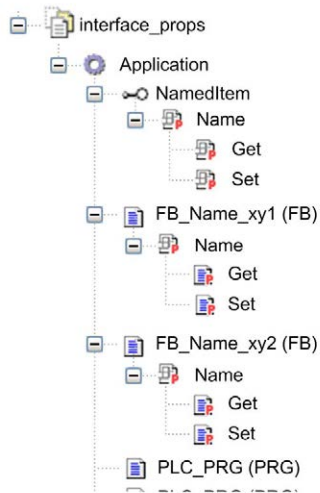
An interface property extends the description of an interface. Such as the interface, it just defines that the accessor methods `Get` and/or `Set` (you can use both or just one of them) belong to the interface; however, provides no implementation code for them. When a function block is extended with an interface containing properties, these properties and their associated `Get` and/or `Set` accessors are automatically inserted in the **Devices tree** below the function block object. They can be edited in order to add the desired implementation code.

Example

In the following figure, the interface `NamedItem` has got a property `Name` with a `Get` and a `Set` accessor method. The `Get` accessor in this example is intended to be used for reading the name of any item from a function block implementing the interface. The `Set` accessor can be used to write a name to this function block. Both methods cannot be edited within the interface definition, but later in the function block.

The function block `FB_Name_xy1` has been added to the **Devices tree**, implementing the interface (`FUNCTION_BLOCK FB_Name_xy1 IMPLEMENTS NamedItem`). Therefore, the property `Name` with the `Get` and `Set` methods has been inserted automatically below `FB_Name_xy1`. Here you can edit the accessor methods, for example in a way that variable `name_of_xy1` is read and thus the name of an item is got. In another function block, also implementing the same interface, the `Get` method can be filled with another code. This code can provide the name of any other item. The `Set` method in the example is used to write a name - defined by program `PLC_PRG ( 'abc' )` - to the function block `FB_Name_xy2`.

Interface NamedItem implemented in 2 function blocks



2 Function Blocks Implementing the Interface NamedItem

Function block FB_Name_xy1

```

FUNCTION_BLOCK FB_Name_xy1 IMPLEMENTS NamedItem
VAR_INPUT
END_VAR
VAR_OUTPUT
END_VAR
VAR
    name_of_xy1: STRING:='xy1';
END_VAR
  
```

Function block FB_Name_xy2

```

FUNCTION_BLOCK FB_Name_xy2 IMPLEMENTS NamedItem
VAR_INPUT
END_VAR
VAR_OUTPUT
END_VAR
VAR
    name_of_xy2: STRING:='xy2';
    name_got_from_PLC_PRG: STRING;
END_VAR
  
```

Implementation of Code in the Accessor Methods Get and Set Below the 2 Function Blocks

FB_Name_xy1.Get

```
VAR
END_VAR
name := name_of_xy1;
```

FB_Name_xy2.Get

```
VAR
END_VAR
name := name_of_xy2;
```

FB_Name_xy2.Set

```
VAR
END_VAR
name_got_from_PLC_PRG:=name;
```

Accessing the Function Blocks by Program PLC_PRG

```
PROGRAM PLC_PRG
VAR
    FBxy1_inst: FB_Name_xy1;
    FBxy2_inst: FB_Name_xy2;
    namexy1: STRING;
    namexy2: STRING;
END_VAR
//get name out of fb
namexy1:=FBxy1_inst.Name;
namexy2:=FBxy2_inst.Name;
//write name to fb
FBxy2_inst.Name:='abc';
```

Action

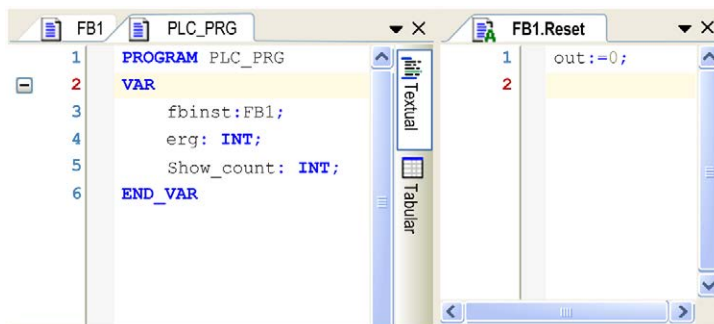
Overview

You can define actions and assign them to function blocks (*see page 181*) and programs (*see page 160*). 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**.

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:

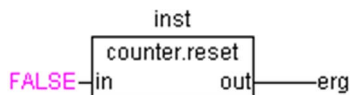
```
CAL Inst.Reset(In := FALSE)
LD Inst.out
ST ERG
```

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.

External Function, Function Block, Method

Overview

For an external function, function block, or method no code will be generated by the programming system.

Perform the following steps to create an external POU:

Step	Action
1.	Add the desired POU object to the Global node of the Applications tree of your project such as any internal object and define the respective input and output variables. NOTE: Define local variables in external function blocks. Do not define them in external functions or methods. VAR_STAT variables cannot be used in the runtime system.
2.	Define the POU to be external: For this purpose, right-click the POU object in the Global node of the Applications tree and execute the command Properties from the context menu. Open the Build tab and activate the option External Implementation (Late link in the runtime system).

In the runtime system an equivalent function, function block or method has to be implemented. At a program download, the equivalent for each external POU is searched in the runtime system. If the equivalent is found, it is linked.

POUs for Implicit Checks

Overview

Below an application you can add special POU's. They have to be available, if the implicitly provided check functionality for array and range boundaries, divisions by zero and pointers during runtime should be used. You can deactivate this functionality in case of devices for which those check functions are provided by a special implicit library.

For this purpose, the **Add Object** → **POUs for implicit checks** dialog box provides the following functions:

- CheckBounds (*see page 614*)
- CheckDivInt (*see page 647*)
- CheckDivLInt (*see page 647*)
- CheckDivReal (*see page 647*)
- CheckDivLreal (*see page 647*)
- CheckRange (*see page 621*)
- CheckRangeUnsigned (*see page 621*)
- CheckPointer (*see page 604*)

After you have inserted a check POU, it is opened in the editor corresponding to the implementation language selected. A default implementation you can adapt to your requirements is available in the ST editor.

After you have inserted a certain check POU, the option is no longer available in the dialog box thus avoiding a double insertion. If all types of check POU's have already been added below the application, the **Add Object** dialog box does not provide the **POUs for implicit checks** option any longer.

CAUTION

INCORRECT IMPLICIT CHECKS 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: As from SoMachine V4.0 after having removed implicit check function (such as CheckBounds) from your application, no **Online Change** is possible, just a download. An appropriate message will appear.

Section 7.2

Function Block

What Is in This Section?

This section contains the following topics:

Topic	Page
General Information	182
Function Block Instance	185
Calling a Function Block	186
Extension of a Function Block	188
Implementing Interfaces	191
Method Invocation	193
SUPER Pointer	195
THIS Pointer	197

General Information

Overview

A function block is a POU (*see page 155*) 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 (*see page 188*) of other function blocks. They can include interface (*see page 191*) definitions concerning Method invocation (*see page 193*). Therefore, inheritance can be used when programming with function blocks.

A function block always is called via an instance (*see page 185*), 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 **Software Catalog** → **Assets** or **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** or **Assets**.

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 <i>Extension of a Function Block</i> (see page 188).
Implements	Enter the names of interfaces (see page 170) 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 <i>Implementing Interfaces</i> (see page 191).
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 SoMachine/CoDeSys compiler version mapping table in the SoMachine Compatibility and Migration User Guide (see SoMachine Compatibility and Migration, User Guide).
Method implementation language	Choose the desired programming language for all method 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>
```

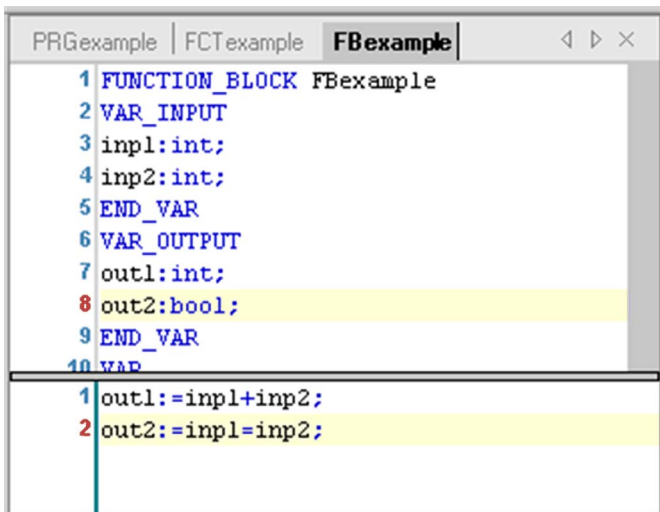
This is followed by the declaration of the variables.

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

A screenshot of a software interface showing the definition of a function block named 'FBexample'. The interface has a tabbed window with three tabs: 'PRGexample', 'FCTexample', and 'FBexample', with 'FBexample' being the active tab. The code is written in a syntax-highlighted ST (Structured Text) language. It starts with 'FUNCTION_BLOCK FBexample', followed by a 'VAR_INPUT' section with two integer variables 'inp1' and 'inp2'. Then, an 'END_VAR' statement is followed by a 'VAR_OUTPUT' section with an integer variable 'out1' and a boolean variable 'out2'. Another 'END_VAR' statement follows. The final line is 'END FBexample'. The code is divided into two sections by a horizontal line. The first section contains the variable declarations, and the second section contains the logic for the function block, which includes 'out1:=inp1+inp2;' and 'out2:=inp1=inp2;'. The lines for 'out2' are highlighted in yellow.

```
1 FUNCTION_BLOCK FBexample
2 VAR_INPUT
3   inp1:int;
4   inp2:int;
5 END_VAR
6 VAR_OUTPUT
7   out1:int;
8   out2:bool;
9 END_VAR
10 END FBexample

1 out1:=inp1+inp2;
2 out2:=inp1=inp2;
```

Function Block Instance

Overview

Function blocks are called (*see page 186*) through an instance which is a reproduction (copy) of a function block (*see page 182*).

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. But instance declarations are not permitted in functions.

Calling a Function Block

Overview

Function blocks (*see page 182*) 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* (*see page 185*) 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 (*see page 155*) 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.

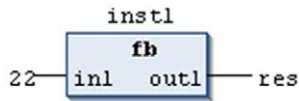
NOTE: If at least 1 of the function block variables is a remanent variable, the total instance is stored in the retain data area.

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 v
value 22 *)
inst1(); (* fb is called, this is needed for the following access on th
e 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 ([see page 527](#)) 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 := 300);
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);
```

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 properties 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.

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 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 use 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 (*see page 195*) (**SUPER**^.<method>).

NOTE: Multiple inheritance is not allowed.

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 (*see page 170*) which allows you to use methods (*see page 165*).

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 property (*see page 174*)) 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. Execute the command Implement interfaces... (*see SoMachine, Menu Commands, Online Help*) to perform this update explicitly.

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 (*see page 188*) by using the keyword EXTENDS in the declaration.

Method Invocation

Overview

Object-oriented programming with function blocks is - besides of the possibility of extension ([see page 188](#)) via EXTENDS - supported by the possible use of interfaces ([see page 191](#)) 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 basic 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* ([see page 165](#)) for further information on methods.
- *THIS Pointer* ([see page 197](#)) for using THIS pointer.
- *SUPER Pointer* ([see page 195](#)) for using SUPER pointer.

Calling Methods

According to the IEC 61131-3 standard, methods such as normal functions ([see page 162](#)) can have additional outputs. They have to 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 output 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 c
alled, else fub1.method1 is called *)
instRef.method1();              (* If b is true, fub1.method1 is call
ed, 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* ([see page 197](#)).

SUPER Pointer

Overview

For each function block, a pointer with name SUPER is automatically available. It points to the basic function block instances, from which the function block is created with inheritance of the basic function block.

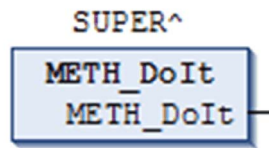
This provides an effective solution for the following issue:

- SUPER offers access to methods of the base class 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 basic function block, you have to unreference it to get the address of the function block: `SUPER^.METH_DoIt`

SUPER Call in Different Implementation Languages

Implementation Language	Example
ST	<code>SUPER^.METH_DoIt();</code>
FBD/CFC/LD	

NOTE: The functionality of SUPER is not yet implemented for Instruction List.

Example

Local variable iVarB overload the function block variable iVarB.

```
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;

END_VAR
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:

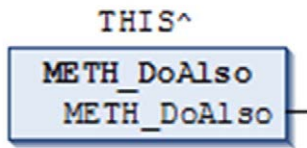
- If a locally declared variable in the method hides a function block variable.
- If you want to refer a pointer to its own function block instance for using in a function.

THIS may only be used in methods 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 inheriting function block, you have to unreference it to get the address of this overriding function block: `THIS^.METHODDoIt`.

THIS Call in Different Implementation Languages

Implementation Language	Example
ST	<code>THIS^.METHODDoIt();</code>
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();

```

Section 7.3

Application Objects

What Is in This Section?

This section contains the following topics:

Topic	Page
Data Type Unit (DUT)	201
Global Variable List - GVL	203
Global Network Variable List - GNVL	205
Persistent Variables	213
External File	214
Text List	216
Image Pool	223

Data Type Unit (DUT)

Overview

Along with the standard data types, you can define your own data types. You can create structures (*see page 617*), enumeration types (*see page 619*), and references (*see page 607*) as data type units (DUTs) in a DUT editor (*see page 391*).

For a description of the particular standard and the user-defined data types, refer to the description of the data types (*see page 600*).

Adding a DUT Object

To add a DUT object to an existing application, select the application node in the **Software catalog** → **Assets** or 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 **Assets** or **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**.

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. This means that 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 (*see page 617*), or an enumeration (*see page 619*).

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 ([see page 537](#)). If a GVL is placed in the **Global** node of the **Software catalog** → **Assets** → **POUs** or 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 **Software catalog** → **Assets** → **POUs** or **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 ([see page 393](#)) to edit a global variable list.

The variables contained in a GVL can be defined to be available as network variables ([see page 821](#)) for a broadcast data exchange with other devices in the network. For this purpose, configure appropriate network properties (by default in the menu **View** → **Properties** → **Network 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 POU's.

GVL for Configurable Constants (Parameter List) in Libraries

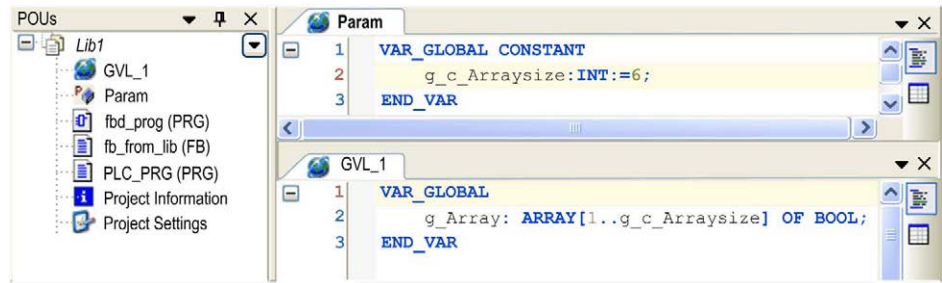
The value of a global constant provided via a library can be replaced by a value defined by the application. For this purpose, the constant has to be declared in a parameter list in the library. Then, when the library is included in the application, its value can be edited in the **Parameter List** tab of the **Library Manager** of the application. See the following example for a description on how to do in detail.

Parameter List Handling

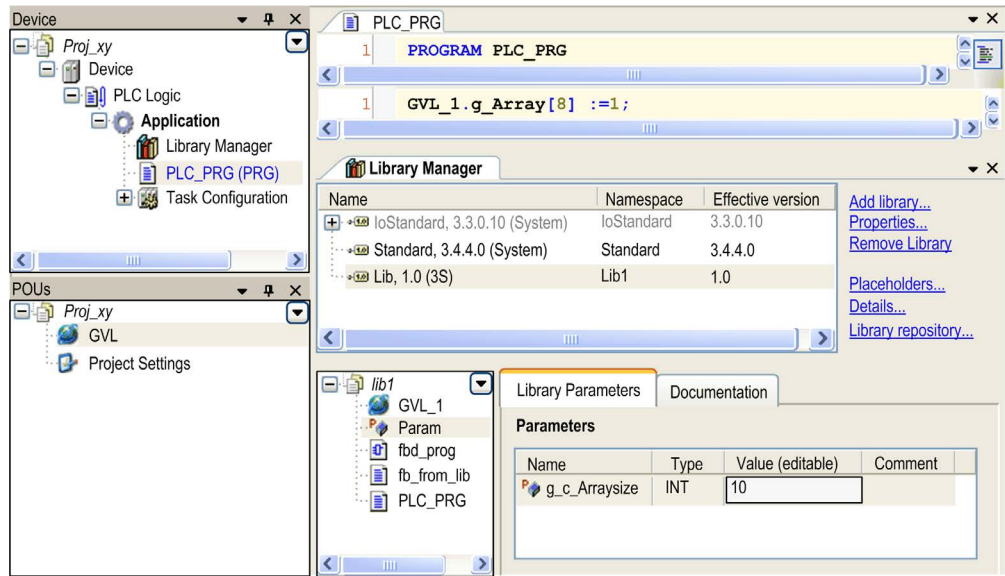
A library `lib1.library` provides an array variable `g_Array`. The size of the array variable is defined by a global constant `g_c_Arraysize`. The library is included in various applications, each needing a different array size. Therefore, you want to overwrite the global constant of the library by an 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 Network Variable List - GNVL

Overview

The **GNVL** functionality is only available if selected in the currently used feature set (**Options → Features → Predefined feature sets**).

A global network variable list (GNVL) is used in the **Software catalog → Variables → Global Variables** view and in the **Applications tree**. It defines variables, which are specified as network variables in another device within the network.

NOTE: The maximum size of a network variable is 255 bytes. The number of network variables is not limited.

Thus you can add a GNVL object to an application if a GVL (*see page 203*) with special network properties (network variable list) is available in 1 of the other network devices. This is independent of whether defined in the same project or in different projects. If several of appropriate GVLs are found within the current project for the current network, choose the desired GVL from a selection list **Sender** when adding a GNVL via the dialog box **Add Object → Add Global Network Variable List**. GVLs from other projects must be imported as described in this chapter.

This means that each GNVL in the current device (receiver) corresponds exactly to 1 GVL in another device (sender).

Dialog box **Add Global Network Variable List**

Add Global Network Variable List

Create a new global network variable list

Name:

Task:

Sender:

Import from file: ...

Description of the Elements

When adding the GNVL, besides a **Name**, also define a **Task**, responsible for the handling of the network variables.

Alternatively to directly choosing a **Sender** GVL from another device, you can specify a GVL export file *.GVL with the option **Import from file**. This GVL file has been generated previously from that **Sender** GVL via **View → Properties → Link To File** dialog box (see *SoMachine, Menu Commands, Online Help*). In any case this is necessary if the desired GVL 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 use the standard 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 *SoMachine, Menu Commands, Online Help*).

A GNVL is displayed by the NVL editor (see page 396), but it cannot be modified. It shows the current content of the corresponding GVL. If you change the basic GVL, the GNVL is updated accordingly.

A comment is added automatically at top of the declaration part of a GNVL, providing information on the sender (device path), the GVL name, and the protocol type.

Global Network Variable List Example

Global network variable list

```

1  //This gobal variable list is received via the network.
2  //Sender: GVL321 [Device_B: Plc Logic: Application]
3  //Protocol: UDP
4
5  VAR_GLOBAL
6      iglobvar321:INT;
7      bglobvar321:BOOL;
8      strglobvar321:STRING;
9  END_VAR

```

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 (see page 821).

Example of a Simple Network Variable Exchange

In the following example, a simple network variable exchange is established. In the sender controller, a global variable list (GVL) is created. In the receiver controller, the corresponding global network variable list (GNVL) 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**:

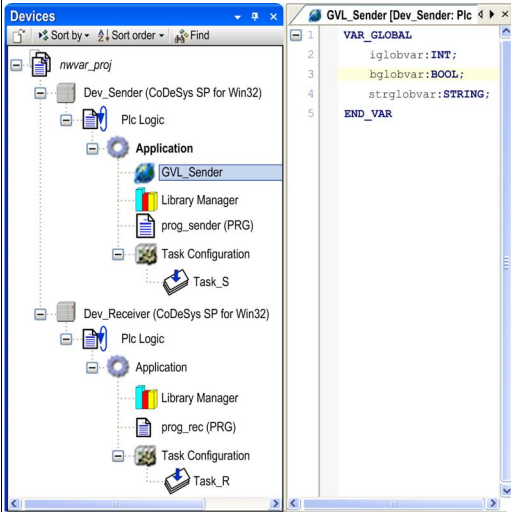
- Create a POU (program) **prog_sender** below the **Application** node of **Dev_Sender**.
- Under the **Task Configuration** node of this application, add the task **Task_S** that calls **prog_sender**.
- Create a POU (program) **prog_rec** below the **Application** node of **Dev_Receiver**.
- Under the **Task Configuration** node of this application, add the task **Task_R** that calls **prog_rec**.

NOTE: The 2 controllers must be configured in the same subnet of the Ethernet network.

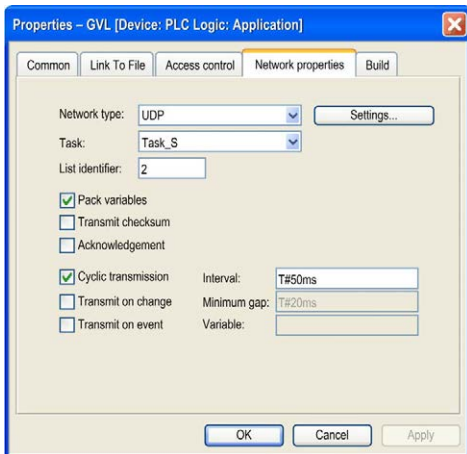
Defining the GVL for the Sender

Step 1: Define a global variable list in the sender controller:

Step	Action	Comment
1	In the Software catalog → Assets View → POU s, select the Application node of the controller Dev_Sender and click the green plus button. Select the command Global Variable List...	The Add Global Variable List dialog box 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 SoMachine screen.

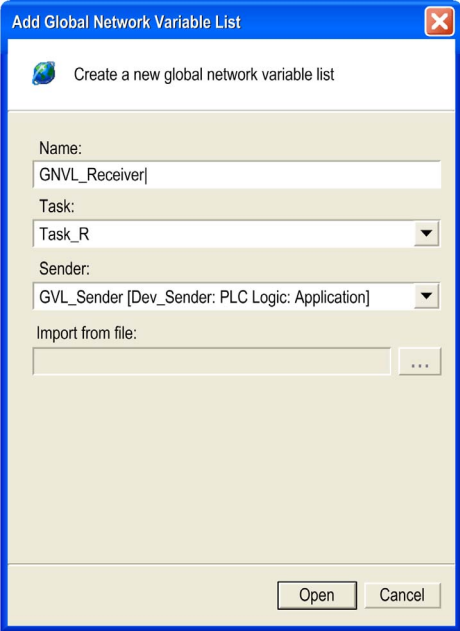
Step	Action	Comment
3	In the editor, enter the following variable definitions: <pre>VAR_GLOBAL iglobvar:INT; bglobvar:BOOL; strglobvar:STRING; END_VAR</pre> 	—

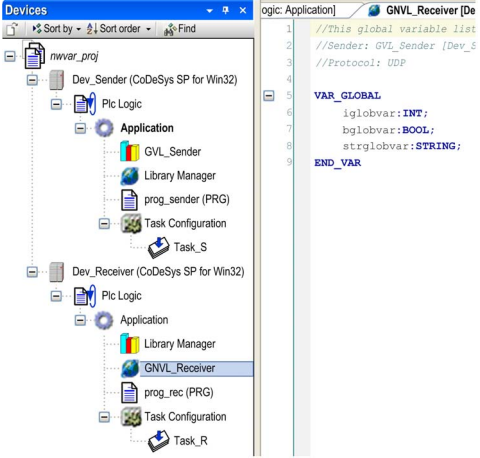
Step 2: Define the network properties of the sender GVL:

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 shown in the graphic: 	–
3	Click OK .	The dialog box is closed and the GVL network properties are set.

Defining the GNVL for the Receiver

Step 1: Define a global network variable list in the receiver controller:

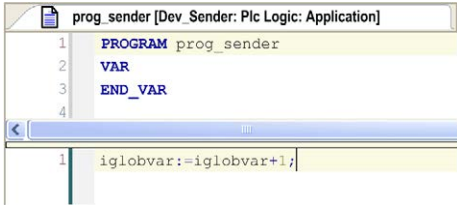
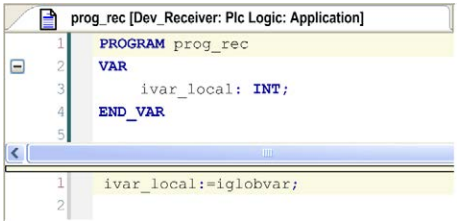
Step	Action	Comment
1	In the Applications tree , select the Application node of the controller Dev_Receiver , click the green plus button, and execute the command Global Network Variable List...	The Add Global Network Variable List dialog box is displayed.
2	Configure the parameters as shown in the graphic. 	This global network variable list is the counterpart of the GVL defined for the sender controller.

Step	Action	Comment
3	Click Open .	The dialog box is closed and the GNVL_Receiver appears below the Application node of the Dev_Receiver controller:  This GNVL automatically contains the same variable declarations as the GVL_Sender.

Step 2: View and / or modify the network settings of the GNVL:

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 shown in the sender.

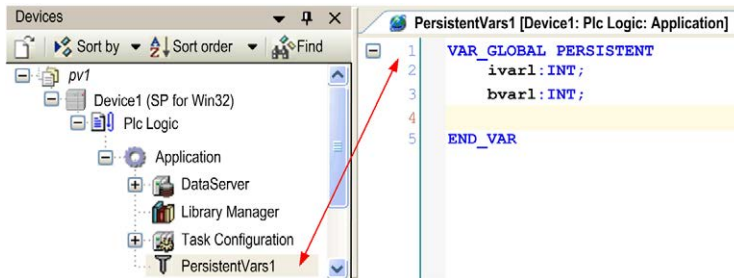
Persistent Variables

Overview

This object is a global variable list, 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....**

Edit a persistent variable list in the GVL editor ([see page 393](#)). The `VAR_GLOBAL PERSISTENT RETAIN` is already preset in the first line.

Persistent variable list



Persistent variables are only reinitialized at a **Reset (origin) <application>**. For further information, refer to the description of remanent variables ([see page 539](#)).

Also refer to the description of the special commands for handling persistent variables ([see SoMachine, Menu Commands, Online Help](#)).

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.

For information on the memory size reserved for retain and persistent variables in the different controllers, refer to the *Programming Guide* of the controller you are using.

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.

Step	Action
3	Click Add . Result: A persistent variable node is created in the Applications tree . For an example, refer to the <i>Overview</i> paragraph in this chapter.

External File

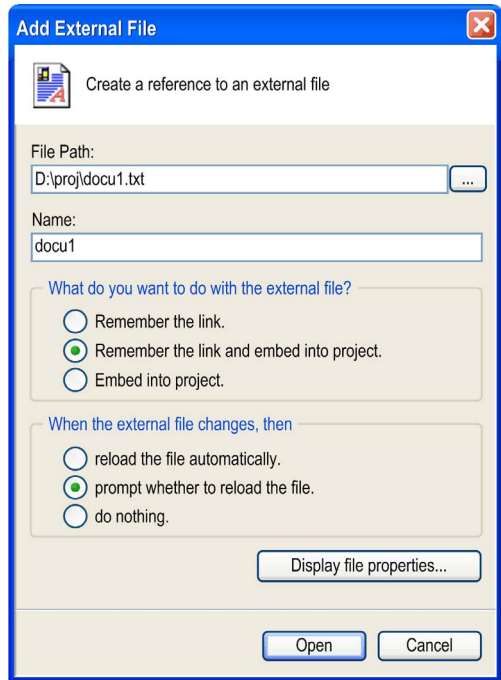
Overview

The **External File** functionality is only available if selected in the currently used feature set (**Options** → **Features** → **Predefined feature sets**).

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....**

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 What Do You Want to Do with the External File? 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.

Description of the When the External File Changes, Then 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 Buttons

Button	Description
Display file properties...	This button opens the standard 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 view and modify the properties.
Open	After you have completed the settings, click the Open button to add the file to the Global node of the Applications Tree or Tools Tree . It is opened in that tool which is defined as default for the given file format.

Text List

Overview

The **Text List** functionality is only available if selected in the currently used feature set (**Options → Features → Predefined feature sets**).

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 ([see page 217](#)) and dynamic ([see page 218](#)) texts and tooltips in visualizations and in the alarm handling
- dynamic text exchange

Text lists can be exported and (re-) imported ([see page 221](#)). Export is necessary, if a language file in XML format has to be provided for a target visualization, but is also useful for translations.

Possible formats of text lists:

- text
- XML

You can activate support of Unicode ([see page 221](#)).

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 language currently set in SoMachine, the default language entry of the text list is used. Each text can contain formatting definitions ([see page 221](#)).

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. Additionally, the list cannot be deleted. However, 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.

Example of a GlobalTextList

Create Global Text List

GlobalTextList			
ID	Standard	Deutsch	Englisch
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 changed 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 currently 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 current text can be changed 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



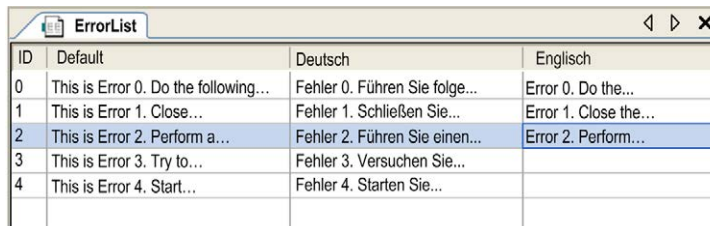
The screenshot shows the SIMATIC Manager project tree on the left with 'ErrorList' selected. The main window displays the 'ErrorList' table configuration with the following data:

ID	Default	Deutsch	Englisch
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**:



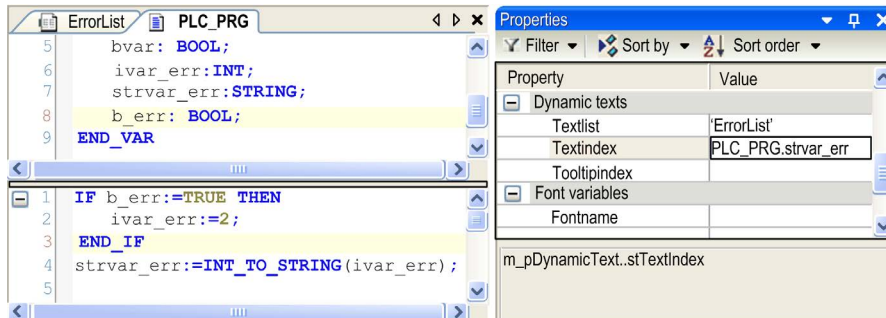
The screenshot shows the 'ErrorList' table configuration with the following data:

ID	Default	Deutsch	Englisch
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...	

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:



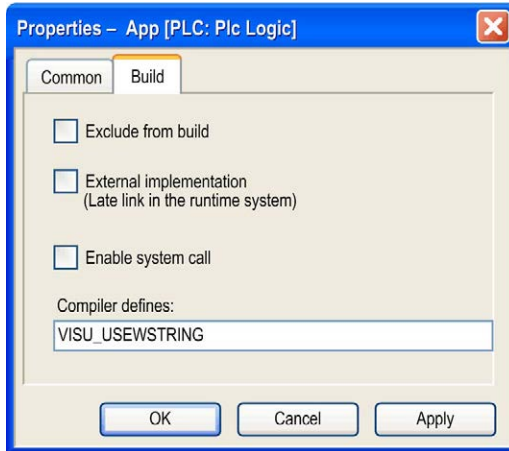
Creating a Text List

- To create a text list for dynamic texts (*see page 218*), 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 (*see page 217*) (**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 SoMachine, 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 current values of variables in a text. For the possible formatting strings, see the *Visualization* part of the SoMachine 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

Via the dialog box **Tools** → **Options** → **Visualization**, you can 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** ([see page 217](#))). 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 shown in the column header of the respective textlist.

Image Pool

Overview

The **Image Pool** functionality is only available if selected in the currently used feature set (**Options** → **Features** → **Predefined feature sets**).

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* ([see page 224](#))).

NOTE: It is recommended to reduce the size of an image file as far 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

ID	File name	Image
drive_icon	C:\Programme\images\SM_Drive.ico	
base_icon	C:\Programme\images\logo.bmp	
12	C:\Programme\images\INTERPOLATO...	

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, <i>C:\programs\images\logo.bmp</i>)
Image	preview of the image

Creating and Editing an Image Pool

A project can contain several image pools.

If a pool is not yet available in a project, then - as soon as you add the first image element and enter an ID (static ID) for the respective image in the visual element properties - an image pool with the default name **GloballImagePool** is created automatically. An entry for the image is inserted.

GloballImagePool is a global pool which is searched first when an image file is to be used. Besides this pool, additional individually named pools can be used.

To create image pools manually, proceed as follows: **GloballImagePool** is created via the command **Image Pool Editor Commands → Create Global Image Pool**. You can insert the other pool objects 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.

To add an image manually to a pool, either use the command **Insert Image**, or fill the pool table manually. For the latter, select the ID field of the first empty line in the pool table, press the SPACE key to open an edit frame, and enter an ID (string). The ID will automatically be made unique. Then set the cursor to the **File** name field, press the SPACE key, and click the ... button to open the **Select image** dialog box. Here you can specify the path of the desired image file.

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 **GloballImagePool**, you do not need to specify the pool name. The search order for images corresponds to that for global variables:
 1. **GloballImagePool**
 2. image pools assigned to the currently active application
 3. image pools in **Global** node of the **Applications tree** besides **GloballImagePool**
 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.

Section 7.4

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 POU's.

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 POU's). 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 and children. For the rules, refer to the description of arranging and configuring objects in the **Devices tree** ([see page 41](#)).

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 SoMachine? Appropriate 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 ([see page 236](#)) for more details.

Chapter 8

Task Configuration

What Is in This Chapter?

This chapter contains the following topics:

Topic	Page
Task Configuration	228
Adding Tasks	229

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 (*see page 225*). 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 Editor** (*see page 419*).

In online mode, the **Task Editor** provides a monitoring view giving 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 context 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 .

Chapter 9

Managing Applications

What Is in This Chapter?

This chapter contains the following sections:

Section	Topic	Page
9.1	General Information	232
9.2	Building and Downloading Applications	234
9.3	Running Applications	249
9.4	Maintaining Applications	250

Section 9.1

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. This means that 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.

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 SoMachine, Menu Commands, Online Help*) to download the boot application to the controller.

Section 9.2

Building and Downloading Applications

What Is in This Section?

This section contains the following topics:

Topic	Page
Building Applications	235
Login	236
Build Process at Changed Applications	238
Downloading an Application	239

Building Applications

Overview

SoMachine provides different build procedures in the Build menu (*see SoMachine, Menu Commands, Online Help*). These procedures serve to handle syntactical checks, either just on the changed objects or on all objects of the active application.

You can perform an offline code generation in order to check for compilation errors before downloading the code to the device. For a successful login, the code generation must have been completed without detecting any errors.

Code Generation, Compile Information

Machine code will not be generated until the Application (*see page 225*) project is downloaded to the target device (controller, simulation target). 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.

No code generation is performed when the project is compiled by the build commands (by default in the **Build** menu). The build process checks the project in order to detect programming errors. Any detected programming errors are displayed in the **Messages** view (message category **Build**).

During code generation, additional errors can be detected and displayed. These errors can only be detected by the code generator or they are caused by memory allocation.

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* ([see page 892](#))) established on the target device, at login you are prompted to enter the appropriate user name and password. For this purpose, the dialog box **Device User Logon** opens.

Login Procedures

2 different login procedures are available, depending on the **Dial-up** mode selected in the **Project** → **Project Settings** → **Communication settings** dialog box. The default setting of this **Dial-up** mode depends on the SoMachine version. The device editor provides individual dialogs for each login type.

Login type	Dial-up mode	Default setting for SoMachine version	Dialog of the device editor
1	IP address	V4.0 and later	Controller Selection (see page 99)
2	active path	V3.1 and earlier	Communication Settings (see page 114)

Login Procedure with Dial-up Mode IP Address

This is the default login procedure for SoMachine V4.0 and later versions. The **Dial-up** mode in the **Project** → **Project Settings** → **Communication settings** dialog box is set to **Dial up via “IP-address”**.

For a successful login, the code generation must have been completed without detecting errors (refer to the chapter *Build Process Before Login* ([see 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 Controller selection view of the device editor opens. A message box is displayed indicating that a valid address has not been defined.
2	If only one controller has been detected by SoMachine scanning the Ethernet network, this controller is marked in the list and it is used as target device. If several controllers have been detected, double-click the controller you want to log in.
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, you can download the application (see page 239).

Login Procedure with Dial-up Mode Active Path

This is the default login procedure for SoMachine V3.1 and earlier versions. The **Dial-up** mode in the **Project Settings → Communication settings** dialog box is set to **Dial up via "active path"**.

For a successful login, the code generation must have been completed without detecting errors (refer to the chapter *Build Process Before Login* ([see page 238](#))). Furthermore, the communication settings ([see page 99](#)) of the device must be configured correctly.

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 view of the device editor opens. A message box is displayed indicating that the active path has not been set and that the network is being scanned.
2	If only one controller has been detected by SoMachine scanning the Ethernet network, this controller is marked in the list and it is used as target device. If several controllers have been detected, double-click the controller you want to log in. NOTE: 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, you can download the application (see page 239).

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. This means that 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 SoMachine 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. This means that 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.

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 SoMachine to the controller.
- Case 2: Modifications have been made to the application that is loaded in the controller in comparison to the application in SoMachine. 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.

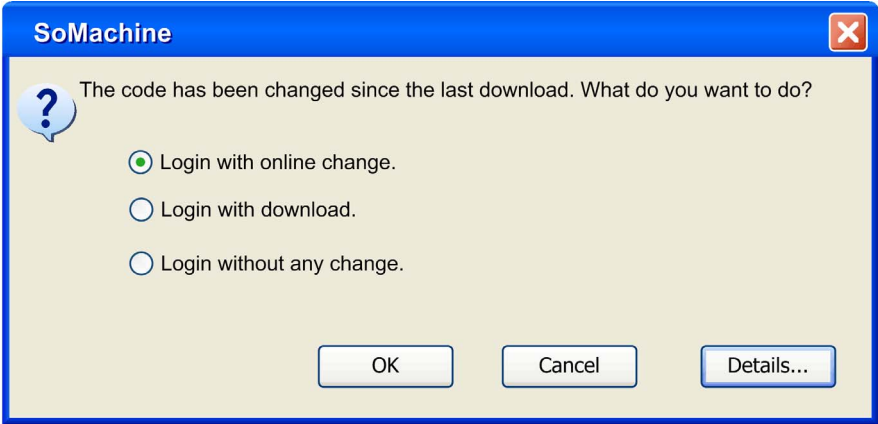
Downloading Your Application to the Controller: Case 1

The application in the controller is the same as the one you want to load. In this case, no download occurs, you just connect SoMachine 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

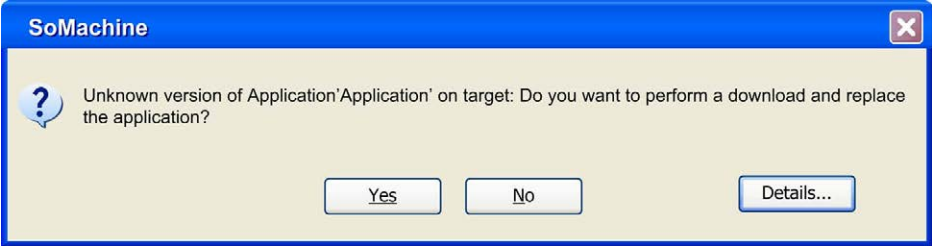
Modifications have been made to the application that is loaded in the controller in comparison to the application in SoMachine.

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 SoMachine application are not downloaded to the controller. In this case, the information and status bar in SoMachine will show RUN 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: See the Programming Guide for 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.

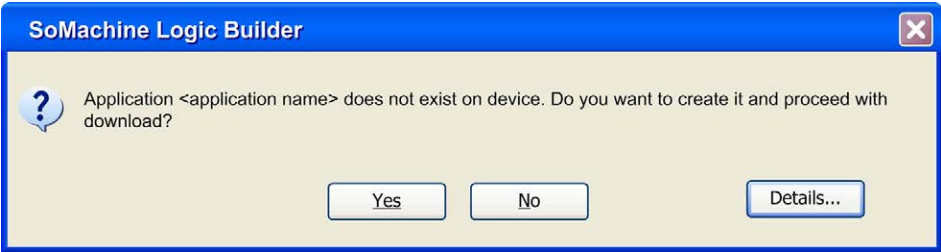
Failure to follow these instructions can result in death, serious injury, or equipment damage.

If you click **Yes**, the application running in your controller will be overwritten.

To help to prevent loss of information, cancel this operation by clicking **No** and execute the **Source Upload** command beforehand. The application currently available on your controller will be loaded to your PC. You can then compare it with the one you intend to download.

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: See the Programming Guide for 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.

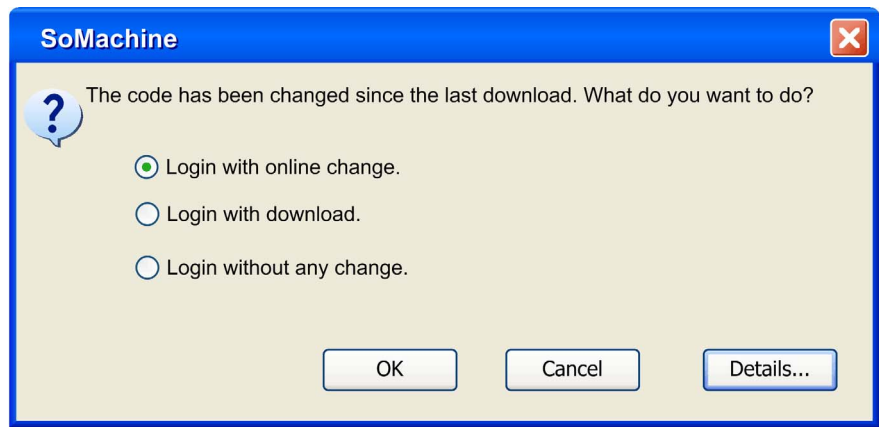
NOTE: For specific information, refer to the chapter *Controller States Description* in the Programming Guide of your controller.

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.

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.

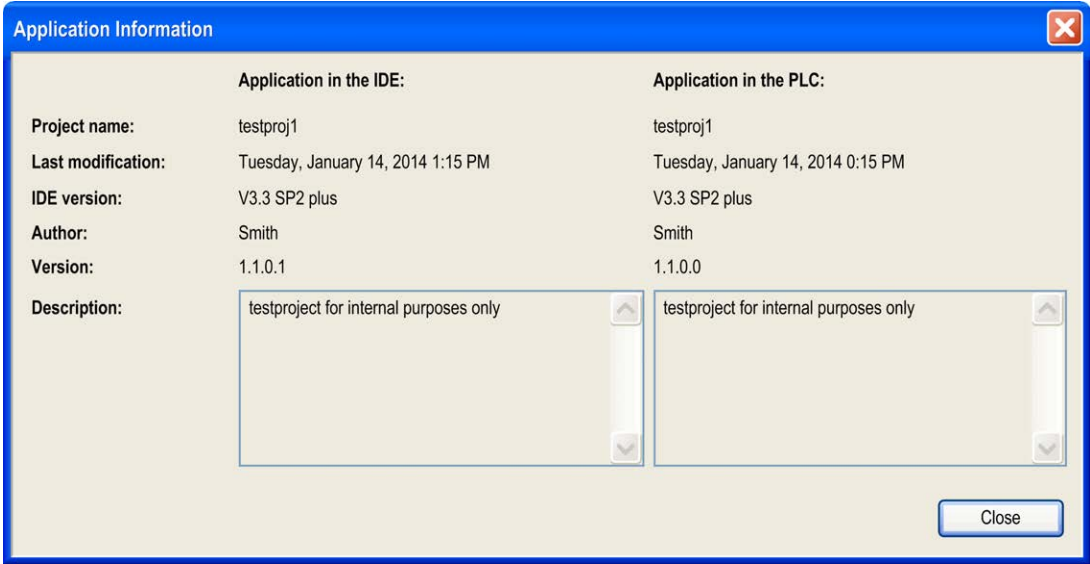
Login dialog box:



Description of the elements:

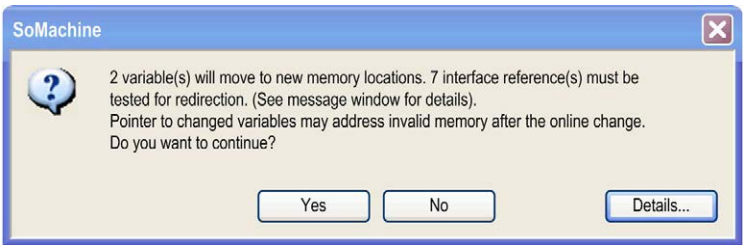
Element	Description
Login with online change	This option is selected per default. If you confirm the dialog box by clicking OK , the modifications will be loaded and immediately displayed in the online view (monitoring) of the respective object or objects.
Login with download	Activate this option to 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.
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., SoMachine) in comparison to that currently available on the controller. Refer to the following figure and to chapter <i>Application Information</i> (see page 235).

Application Information dialog box



For further information, refer to the *Login* chapter (see page 236).

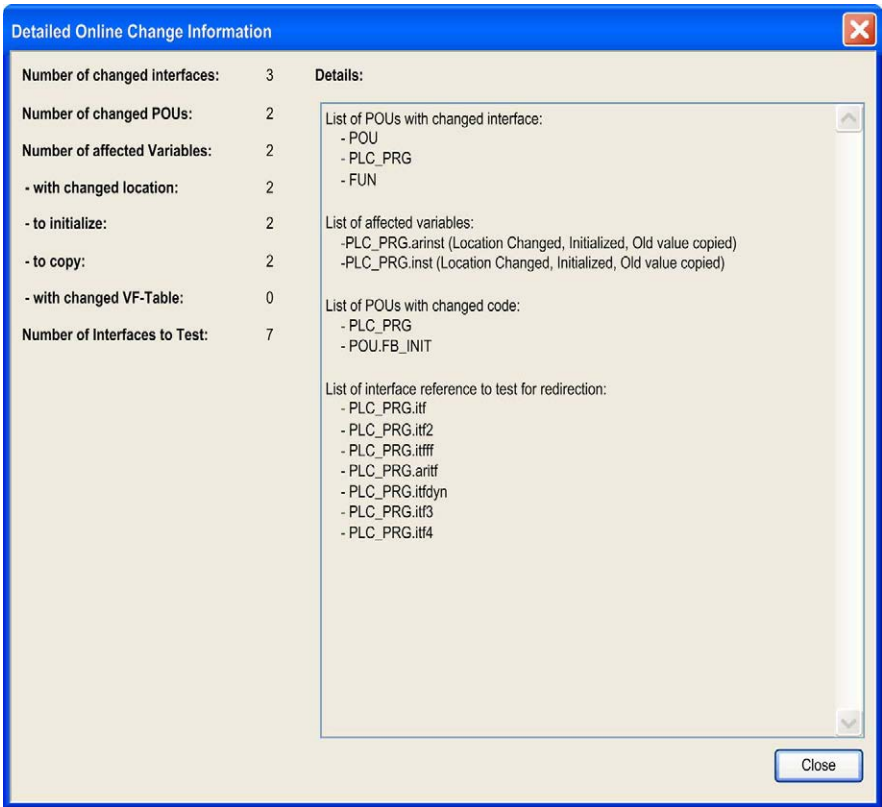
If the online change will affect considerable changes in download code, like for example possible moves of pointer addresses or necessary redirections of interface references (see page 170) another message box is displayed after you have confirmed the **Online change** dialog box with **OK** before download will be performed. It informs you about the effects you have to consider and provides the option to abort the online change operation.



NOTE: With SoMachine V4.0 and later, after having removed implicit check function (such as CheckBounds) from your application, no **Online Change** is possible, just a download. An appropriate 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, POU's, 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 replaced, 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 the file *application.app* in the controller system folder, thus making it available as a boot application. A boot application is the project which 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).

The **Create boot application** command will copy the *application.app* file to a file called *<projectname>.app* and thereby making it the boot application for the controller. You can also create the boot application while in offline mode (*see 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 SoMachine, 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 SoMachine, Functions and Libraries User Guide*)).
2. Ensure that a definitive compiler version is set in the **Project Settings → Compile options** dialog box (*see SoMachine, 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 SoMachine 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 SoMachine, 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 via SoMachine Central. Select the following information: **Download information files, Library profile, 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.

Section 9.3

Running Applications

Running Applications

Introduction

This part shows how to start/stop an application.

RUN/STOP with SoMachine

The controller can be started and stopped using SoMachine run on a PC connected to the controller.

Click **Online** → **Start 'Application [ApplicationName: Plc logic]'** or CTRL + F5 or the **Start 'Application [ApplicationName: Plc logic]'** button in the menu bar to start the application.

Click **Online** → **Stop 'Application [ApplicationName: Plc logic]'** or CTRL +SHIFT + F5 or the **Stop 'Application [ApplicationName: Plc logic]'** button in the menu bar to stop the application.

RUN/STOP Input for Controllers

Some controllers allow you to configure a Run/Stop input to control application start/stop.

Status	Description
0	Stop the application. RUN command in SoMachine is not possible.
Rising Edge	Start the application.
1	The application runs. RUN/STOP command in SoMachine is possible.

Refer to the manual of your controller to find out whether it supports this function.

Configure and use the RUN/STOP input if you are going use remote commands to start and stop the controller. It is the best method to help ensure local control over the running of the controller and to help prevent the inadvertent start of the controller from a remote location.

WARNING

UNINTENDED MACHINE OR PROCESS START-UP

- Verify the state of security of your machine or process environment before applying power to the Run/Stop input.
- Use the Run/Stop input to help prevent the unintentional start-up from a remote location.

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

Section 9.4

Maintaining Applications

What Is in This Section?

This section contains the following topics:

Topic	Page
Monitoring	251
Debugging	252

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 ([see page 381](#)).
- You can view objects independently in lists through the command **Watch**. For details, refer to the description of the watch view / watch list editor ([see page 430](#)). You can insert a variable in a watch view by selecting it and executing the command **Add watchlist** from the context 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 ([see page 466](#)).
- 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 ([see page 447](#)).

For information on monitoring of properties that are inserted beneath POU's or function blocks, refer to the chapter Property ([see page 168](#)).

For information on monitoring of function calls, refer to the chapter *Attribute Monitoring* ([see page 578](#)).

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. For more information, see the Programming Guide for your particular controller.

Debugging

Overview

To evaluate potential programming errors, you can use the debugging functionality in online mode. You can also, to some degree, debug your application in simulation. While simulation avoids the need to connect to physical hardware, there are limitations to which you may need to complete debugging online.

You can set breakpoints at certain positions to force an execution break. Certain conditions, such as which tasks are identified and in which cycles the breakpoint should be effective, can be set for each breakpoint (conditional breakpoints). Refer to the description of *Breakpoints* in this chapter.

Stepping functions are available which allow a program to be executed in controlled steps. Refer to the *Stepping* paragraph ([see page 253](#)).

At each break, you can examine the current values of the variables. You can view a Call Stack (*see SoMachine, Menu Commands, Online Help*) for the current step position.

You can activate the **Flow Control** function in order to track the executed parts of the application program. In contrast to the standard monitoring function which only shows the value of a variable between 2 execution cycles, **Flow Control** provides the value at each particular execution step, exactly at the time of execution.

Breakpoints

A breakpoint that is set in an application program will cause a break during the execution of the program. Only the task having hit the breakpoint (called in the following the debug task) will be halted; however, the others will continue to execute. The possible breakpoint positions depend on the editor. In each case, there is a breakpoint at the end of a POU.

NOTE: The IOs handled by the debug task are not updated on a halt on a breakpoint. This applies even if the option **Update IO while in stop** is enabled in the **PLC settings** tab of the device editor ([see page 126](#)).

Refer to the chapter *Breakpoints Commands* for a description of the commands concerning breakpoints. The **Breakpoints** dialog box (*see SoMachine, Menu Commands, Online Help*) provides an overview on all breakpoints, allowing you to add, remove, and modify breakpoints.

Conditional Breakpoints

The halt on a breakpoint can be made dependent based on the number of cycles of a given task, or on which task is currently active. By declaring a specific debug task you can avoid that every task sharing the same POU be affected but the breakpoint (refer to the *Breakpoints and Stepping in Applications with Multiple Tasks* paragraph ([see page 253](#))).

Breakpoint Symbols

Symbol	Description
●	breakpoint enabled
○	breakpoint disabled
⊙	halt on breakpoint in online mode
➡	current step position indicated by a yellow arrow before the respective line and a yellow shadow behind the concerned operation

Stepping

Stepping allows a controlled execution of an application program, for debugging purposes. 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 *SoMachine, Menu Commands, Online Help*) for a description of the stepping commands.

Example of a Step Into Operation

Starting from the breakpoint you can execute each single command line with the stepping command.

Step Into, example

```
1 ● ldl();
2   erg 0 :=fbinst.ic
3 ➡ IF bvarFALSE THEN
4     ivarl 45 :=23;
5   ELSE
6     ivarl 45 :=45;
7   END IF;
```

Breakpoints and Stepping in Applications with Multiple Tasks

If a breakpoint can be hit by multiple tasks, because the POU is used by multiple tasks, only the task that arrives first will be halted. Consider this in case of single stepping or if you continue debugging after a halt. It could be the case that another task can be halted at the next hit (the cycle is possibly not yet finished). If only 1 certain task should be concerned (debug task), you can specify it in the breakpoint condition properties (**Breakpoints** → **New Breakpoint** dialog box, tab **Condition**).

Part IV

Logic Editors

What Is in This Part?

This part contains the following chapters:

Chapter	Chapter Name	Page
10	FBD/LD/IL Editor	257
11	Continuous Function Chart (CFC) Editor	307
12	Sequential Function Chart (SFC) Editor	327
13	Structured Text (ST) Editor	359

Chapter 10

FBD/LD/IL Editor

What Is in This Chapter?

This chapter contains the following sections:

Section	Topic	Page
10.1	Information on the FBD/LD/IL Editor	258
10.2	FBD/LD/IL Elements	289
10.3	LD Elements	303

Section 10.1

Information on the FBD/LD/IL Editor

What Is in This Section?

This section contains the following topics:

Topic	Page
FBD/LD/IL Editor	259
Function Block Diagram (FBD) Language	260
Ladder Diagram (LD) Language	261
Instruction List (IL) Language	262
Modifiers and Operators in IL	264
Working in the FBD and LD Editor	267
Working in the IL Editor	272
Cursor Positions in FBD, LD, and IL	278
FBD/LD/IL Menu	282
FBD/LD/IL Editor in Online Mode	283

FBD/LD/IL Editor

Overview

A combined editor is available for editing POUs in the languages FBD (function block diagram (*see page 260*)), LD (ladder diagram (*see page 261*)), and IL (instruction list (*see page 262*)).

Therefore, a common set of commands and elements is used and an automatic internal conversion between the 3 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 (*see page 382*), the lower part contains a the coding area.

The programming language for a new object is specified when you create the object.

For further information refer to:

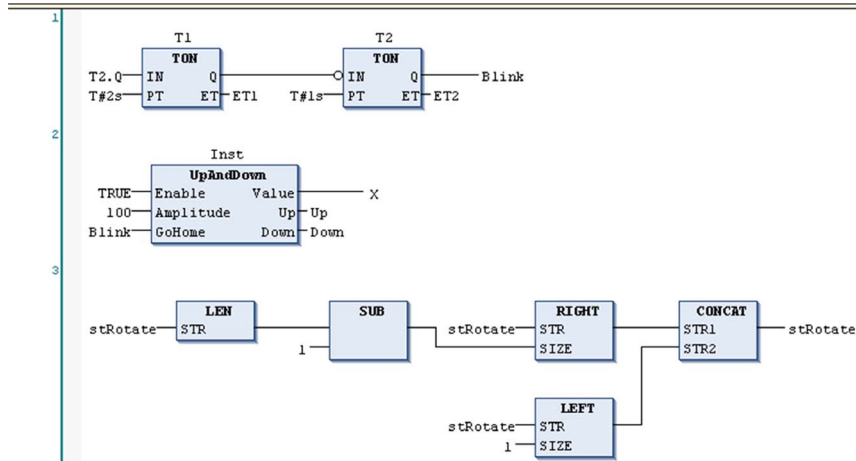
- *Working in the FBD and LD Editor View (see page 267)*
- *Working in the IL Editor View (see page 272)*

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

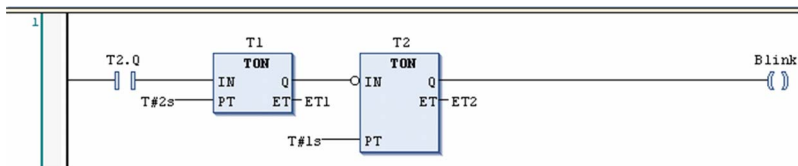
The Ladder Diagram is a graphics-oriented programming language which resembles the structure of an electric circuit.

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 POU's.

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 POU's (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

```
PROGRAM IL
VAR
    inst1: TON;
    dwVar: DWORD;
    dwRes: DWORD;
    t1:    TIME;
    tout1: TIME;
    inst2: TON;
    bVar:  BOOL;
END_VAR
LD      bVar                                variable
ST      inst1.IN                            starts timer with risin...
JMPC    m1
```

```
CAL      inst1(  
          PT:=t1,  
          ET:=>tout1)  
LD        inst1.Q          is TRUE, PT seconds aft...  
ST        inst2.IN         starts timer with risin...  
ml:  
LD        dwVar  
ADD       230  
ST        dwRes
```

For further information, refer to:

- *Working in the IL Editor View* ([see page 272](#))
- *Modifiers and operators in IL* ([see page 264](#))

Modifiers and Operators in IL

Modifiers

You can use the following modifiers in Instruction List (*see page 262*).

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	other operators according to the <i>Operators</i> table below (<i>N</i> in <i>Modifiers</i> column)	Negation of the operand (not of the accumulator).

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, iVar2)

Operator	Modifiers	Meaning	Example
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 or equal to 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
LT	(	Verifies if accumulator is less than operand, result (BOOL) is copied into the accumulator; <	LT cVar1
JMP	CN	Unconditional (conditional) jump to the label	JMPN next
CAL	CN	(Conditional) call of a PROGRAM or FUNCTION_BLOCK (if accumulator is TRUE).	CAL prog1
RET	–	Early return of the POU and jump back to the calling POU	RET
RET	C	Conditional - if accumulator is TRUE,)early return of the POU and jump back to the calling POU	RETC
RET	CN	Conditional - if accumulator is FALSE,)early return of the POU and jump back to the calling POU	RETCN
)	–	Evaluate deferred operation	–

See also IEC operators ([see page 639](#)) and Work in IL editor ([see page 272](#)) 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
```

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.

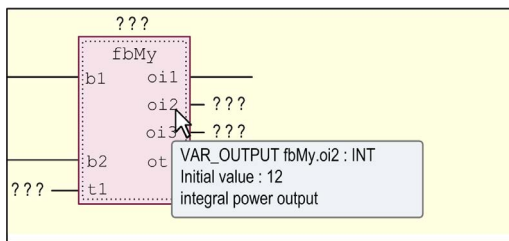
Refer to the general editor settings in the **Options** dialog box, category **FBD/LD/IL** for potential editor display options.

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).

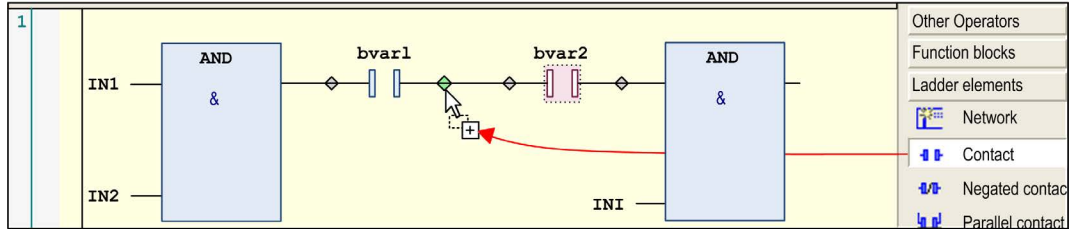
Example: Tooltip on a POU output



Inserting and Arranging Elements

- The commands for working in the editor are by default available in the FBD/LD/IL menu ([see page 282](#)). Frequently used commands are also available in the context menu. It depends on the current cursor position or the current selection (multiselection possible, see below [Selecting](#) ([see page 269](#))) which elements can be inserted via menu command.
- You can also drag elements with the mouse from the toolbox ([see page 290](#)) 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.

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* ([see page 278](#)).
- 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).

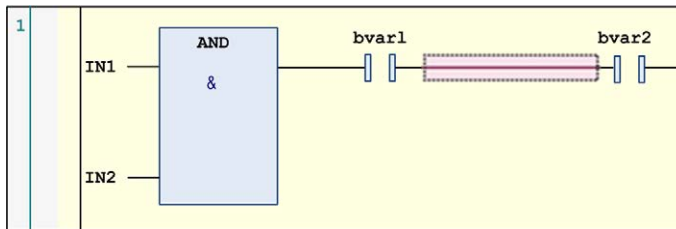
Navigating in the Editor

- You can use the ARROW keys to jump to the next or previous cursor position ([see page 278](#)). 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 ([see page 278](#)) 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* (see page 278).
- 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

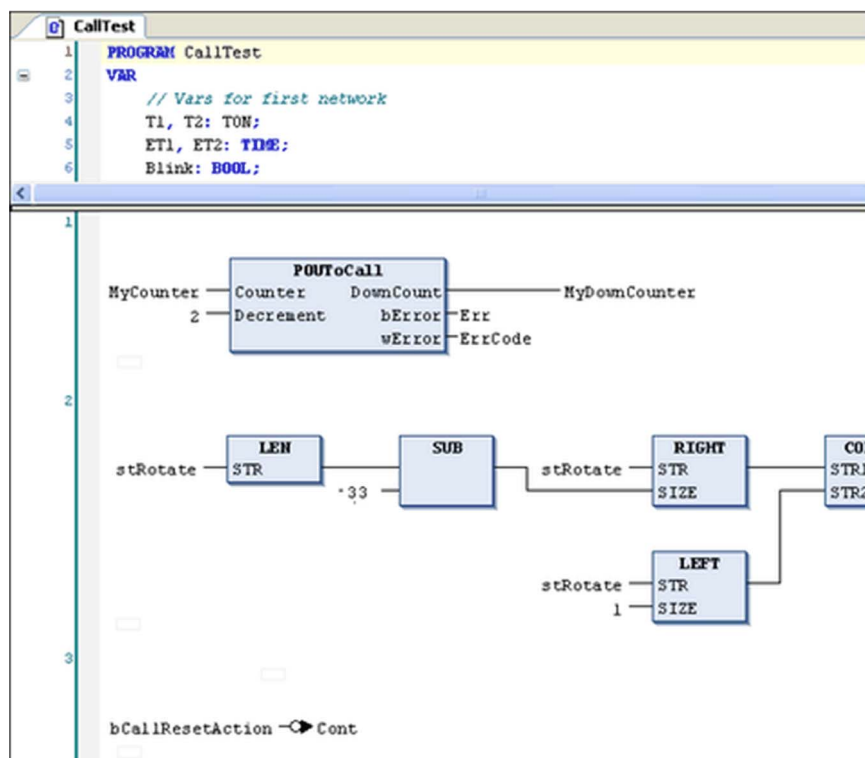


- Multi-selection of elements or networks is possible by keeping pressed the CTRL key while selecting the desired elements one after the other.

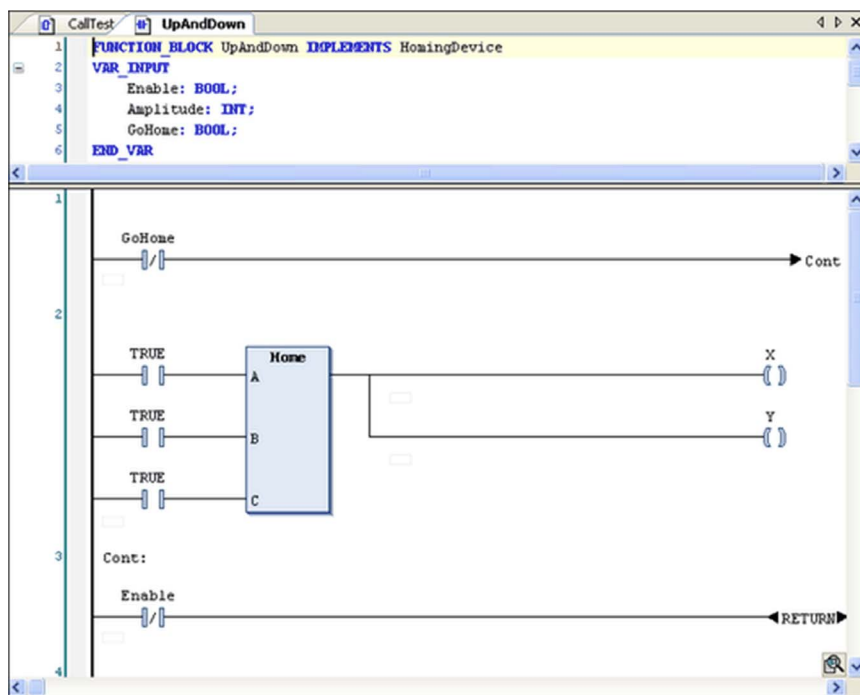
Open a Function Block

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 context menu.

FBD editor



LD editor



For information on the languages, refer to:

- *Function Block Diagram - FBD* (see page 260)
- *Ladder Diagram - LD* (see page 261)

Working in the IL Editor

Overview

The IL Instruction List (*see page 262*) editor is a table editor. The network structure of FBD or LD programs is also represented in an IL program. Basically, one network (*see 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.

Also refer to the general editor settings in the **Options** dialog box, category **FBD/IL/LD**.

Tooltip

Tooltips contain information on variables or box parameters.

Refer to *Working in the FBD and LD Editor View* (*see page 267*).

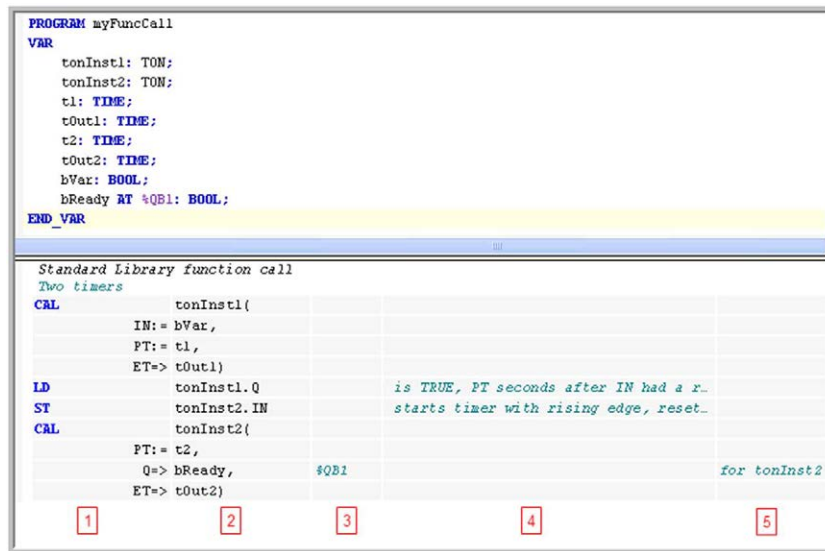
Inserting and Arranging Elements

- The commands for working in the editor are available in the **FBD/LD/IL** menu. Frequently used commands are also available in the context 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 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 (*see page 262*).
- 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:

Column	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 <i>Modifiers and Operators in IL</i> (see page 264).
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 appropriate 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 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 (*see page 264*) 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 (*see page 264*). 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 ([see 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 ([see 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 ([see page 294](#)). The execution of the jump depends on the value loaded.

Example: Conditional jump instruction; in case `bCallResetAction` 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* ([see page 272](#)).

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* ([see page 267](#)).

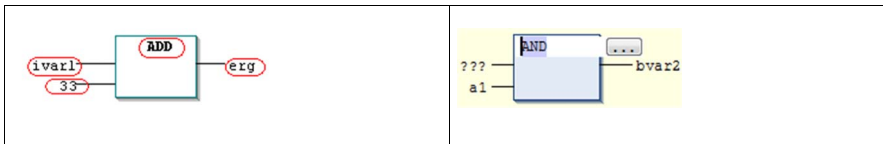
Basically, in FBD a dotted rectangle around the respective element indicates the current position of the cursor. Additionally, texts and boxes become blue- or red-shadowed.

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 context menu for getting inserted ([see page 267](#)).

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:



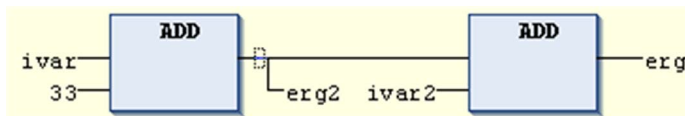
(3) Every operator, function, or function block:



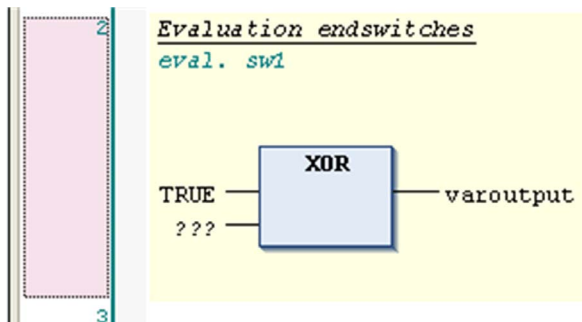
(4) Outputs if an assignment or a jump comes afterward:



(5) Just before the lined cross above an assignment before a jump or a return instruction:



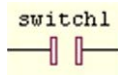
(6) The right-most cursor position in the network or anywhere else in the network besides the other cursor positions. This will select the whole network:



(7) The lined cross directly in front of an assignment:



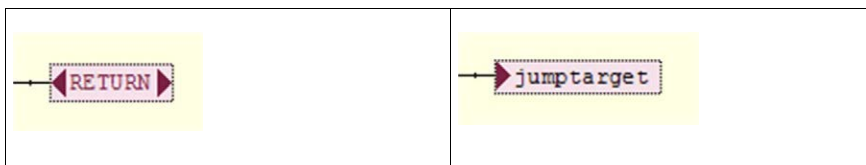
(8) Every contact:



(9) Every coil:



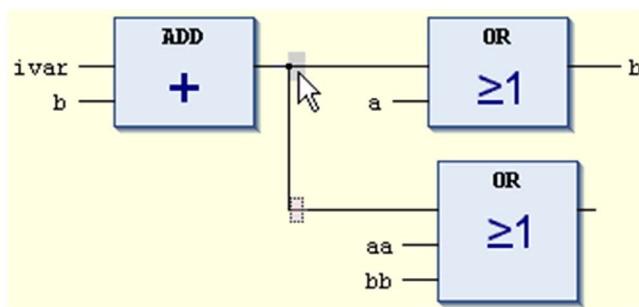
(10) Every return and jump:



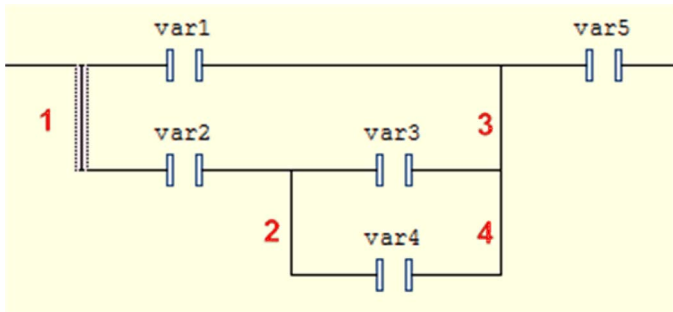
(11) The connecting line between the contacts and the coils:



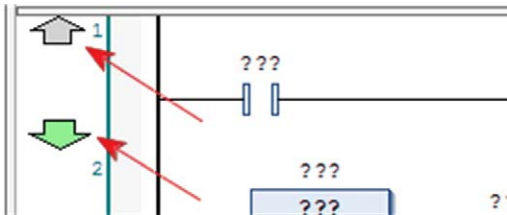
(12) Branch or subnetwork within a network:



(13) The connection line between parallel contacts (Pos. 1...4):

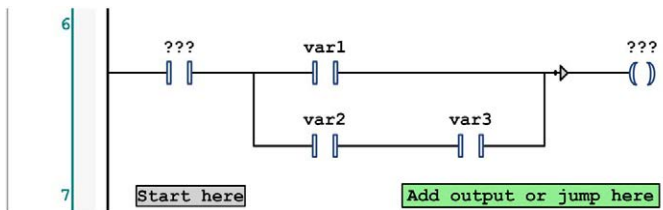


(14) Before or after a network:



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 (*see page 259*) 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:

FBD/LD/IL	Build	Online	Debug	Tools	Win
	Insert Network			Ctrl+I	
	Insert Network (below)			Ctrl+T	
	Insert label				
(x x)	Toggle network comment state			Ctrl+O	
	Insert Box			Ctrl+B	
	Insert Empty Box			Ctrl+Shift+B	
	Insert Box with EN/ENO			Ctrl+Shift+E	
	Insert Empty Box with EN/ENO				
	Insert Input			Ctrl+Q	
-VAR	Insert Assignment			Ctrl+A	
→	Insert Jump			Ctrl+L	
◀RET	Insert Return				
	Negation			Ctrl+N	
	Edge Detection			Ctrl+E	
	Set/Reset			Ctrl+M	
	Set output connection			Ctrl+W	
	Insert Branch			Ctrl+Shift+V	
	Insert Branch above				
	Insert Branch below				
	Update parameters			Ctrl+U	
	Remove unused FB call parameters				
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 (*see page 283*) 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* (*see page 287*).

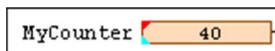
- For information on how to open objects in online mode, refer to the chapter *User Interface in Online Mode* (*see page 51*).
- Keep in mind that the editor window of an FBD, LD, or IL object also includes the **Declaration Editor** in the upper part. Also refer to the chapter *Declaration Editor in Online Mode* (*see page 387*).

Monitoring

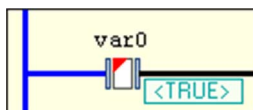
If the inline monitoring is not explicitly deactivated in the **Options** dialog box, it will be supplemented in FBD or LD editor by small monitoring windows behind each variable or by an additional monitoring column showing the actual values (inline monitoring). This is even the case for unassigned 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 (*see page 286*), 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.

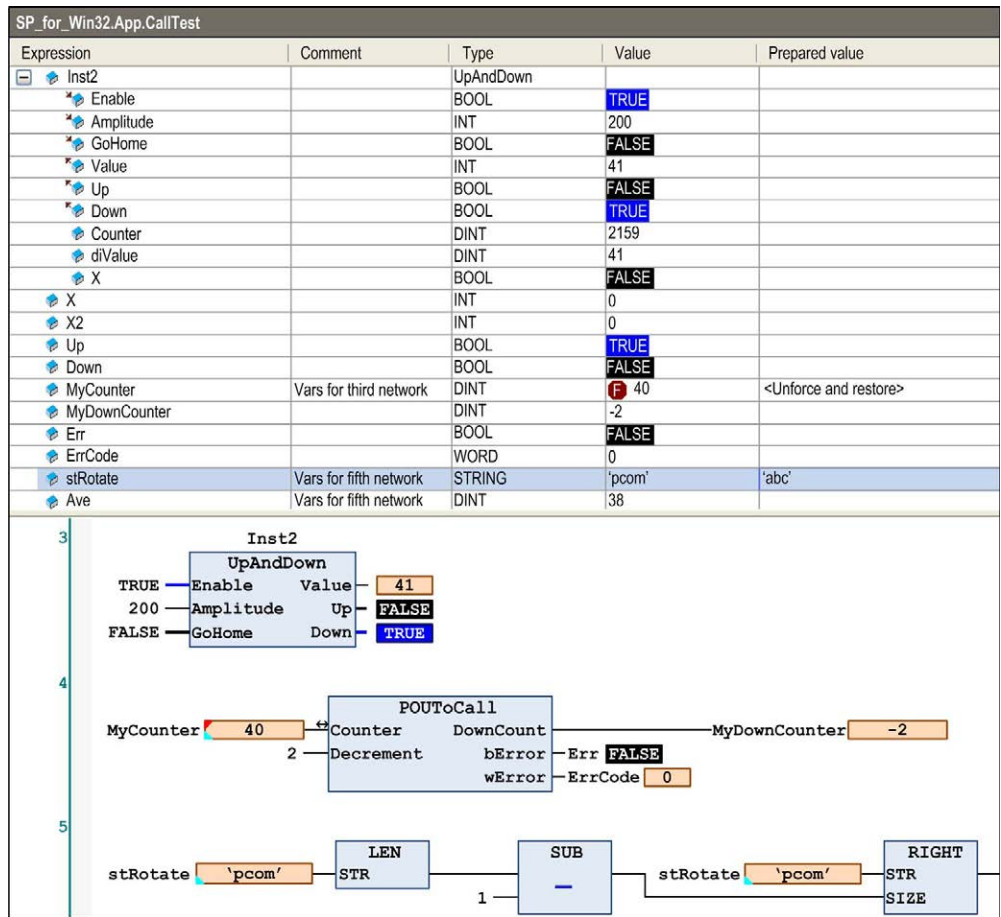
Example for a variable which is currently forced and prepared for releasing the force



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

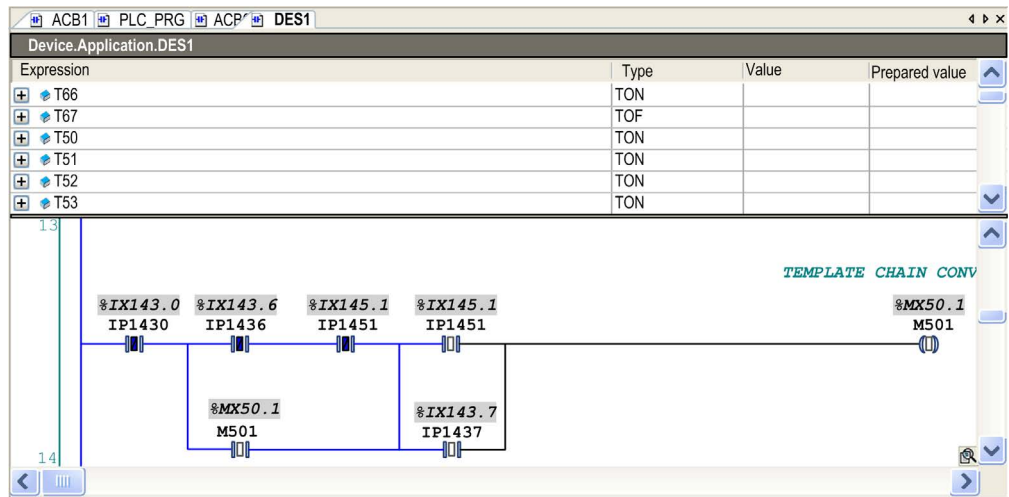
		iVar	
		iRes	
		iTemp	
1	LD	PLC_PRG.iX	11467 ...
	ADD	22	add 22
	ST	iTemp	11488 store to temp
2	LD	PLC_PRG.iX	11467
	ADD	1	
	ST	iVar	11467
3	CAL	PRG1(	call program PRG1
		iIn:= iVar,	11467
		iOut=> iRes)	11468 store iOut to iRes

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 context menu. Refer to the description of the *User Interface in Online Mode* ([see page 51](#)) 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 ([see page 387](#)) 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* ([see page 283](#)).

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:

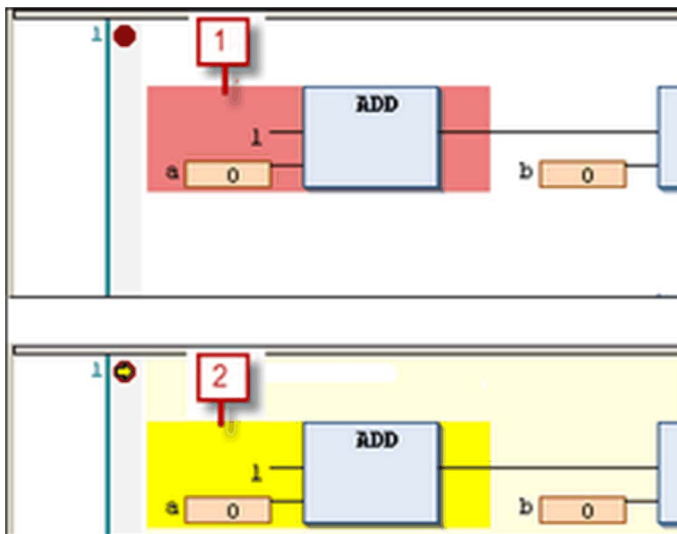
- On the network as a whole such that the breakpoint will be applied to the first possible position within the network.
- On a box (*see page 295*), if this contains a statement. Therefore it is not possible on operator boxes like for example ADD, DIV. See the Note below.
- On an assignment.
- At the 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.

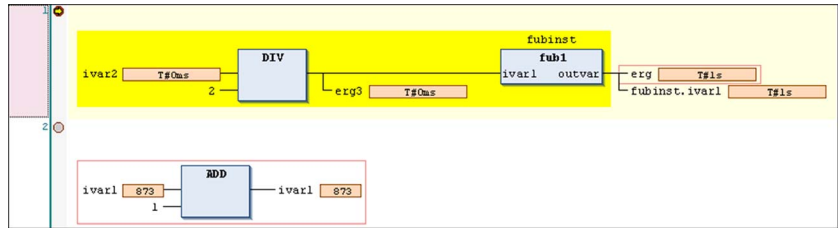
Breakpoint set and breakpoint reached



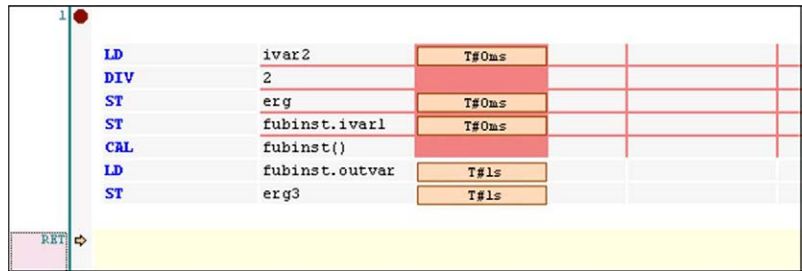
- 1 breakpoint set
- 2 breakpoint reached

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.

Section 10.2

FBD/LD/IL Elements

What Is in This Section?

This section contains the following topics:

Topic	Page
FBD/LD/IL Toolbox	290
Network in FBD/LD/IL	292
Assignment in FBD/LD/IL	294
Jump in FBD/LD/IL	294
Label in FBD/LD/IL	294
Boxes in FBD/LD/IL	295
RETURN Instruction in FBD/LD/IL	295
Branch / Hanging Coil in FBD/LD/IL	296
Parallel Branch	299
Set/Reset in FBD/LD/IL	301
Set/Reset Coil	302

FBD/LD/IL Toolbox

Overview

The FBD/LD/IL Editor (*see page 259*) 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).

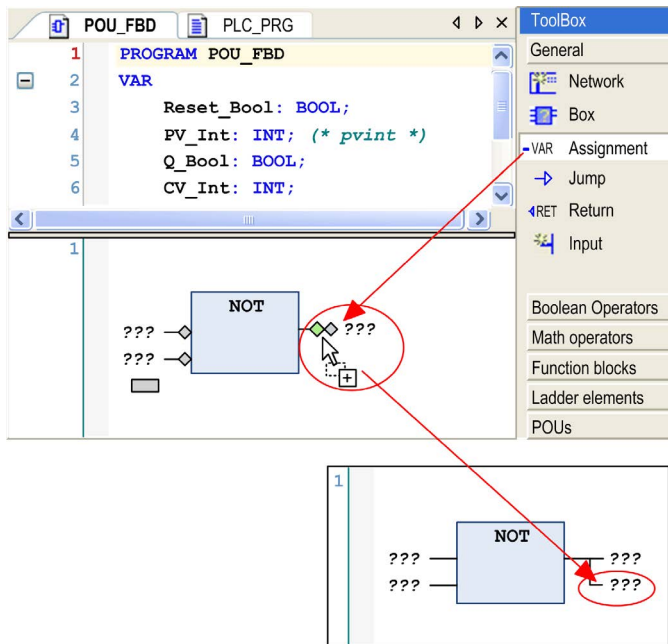
The **POUs** category lists all POU's which have been defined below the same application as the FBD/LD/IL object which is open in the editor. If a POU has been assigned a bitmap in its properties, then this will be displayed before the POU name. Otherwise, the standard icon for indicating the POU type will be used. The list will be updated automatically when POU's 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 appropriate 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 ([see page 260](#)) or LD ([see page 261](#)) 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 ([see page 262](#)), 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.

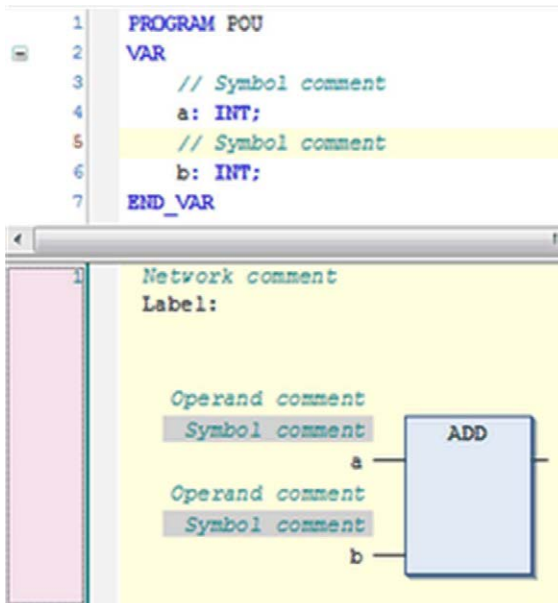
A network optionally can get assigned a title, a comment and a label ([see page 294](#)).

You can switch the availability of the title and the comment fields 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 ([see page 294](#)), which then can be addressed by a jump ([see 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 ([see page 278](#))), you can apply the default commands for copying, cutting, inserting, and deleting.

NOTE: Right-clicking (cursor position 6 ([see page 278](#))) 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 ([see page 290](#)). A network with all belonging elements can also be copied or moved ([see page 267](#)) by drag and drop within the editor.

You can also create subnetworks ([see page 296](#)) 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 ([see page 283](#)).

Assignment in FBD/LD/IL

Overview

Depending on the selected cursor position (*see page 278*) in FBD or LD, an assignment will be inserted directly in front of the selected input (cursor position 2 (*see page 278*)), directly after the selected output (cursor position 4 (*see page 278*)) or at the end of the network (cursor position 6 (*see page 278*)). In an LD network, an assignment will be displayed as a coil (*see page 305*). Alternatively, drag the assignment element from the toolbox (*see page 290*) or copy or move (*see page 267*) 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 (*see page 262*), an assignment is programmed via LD and ST instructions. Refer to *Modifiers and Operators in IL* (*see page 264*).

Jump in FBD/LD/IL

Overview

Depending on the selected cursor position (*see page 278*) in FBD (*see page 260*) or LD (*see page 261*), 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 (*see page 290*) or copy or move (*see page 267*) 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.

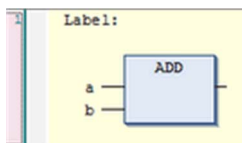
In IL (*see page 262*), a jump is inserted via a JMP instruction. See in this context the description of the operators and modifiers in IL (*see page 264*).

Label in FBD/LD/IL

Overview

Below the network comment field each FBD (*see page 260*), LD (*see page 261*) 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 (*see 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 ([see page 260](#)), LD ([see page 261](#)), or IL ([see page 262](#)) 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 system library or can be programmed. At least 1 input and 1 output however must be assigned to boolean values.

If provided with the respective module and if the option **Show box icon** is activated, an icon will be 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 ([see page 290](#)) 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 ([see page 262](#)) program, a CAL ([see page 264](#)) 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

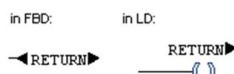
Overview

With a RETURN instruction, the FBD ([see page 260](#)), LD ([see page 261](#)) or IL ([see page 262](#)) 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 ([see page 290](#)) or copy or move ([see page 267](#)) it from another position within the editor.

RETURN element



In IL, the RET ([see page 264](#)) instruction is used for the same purpose.

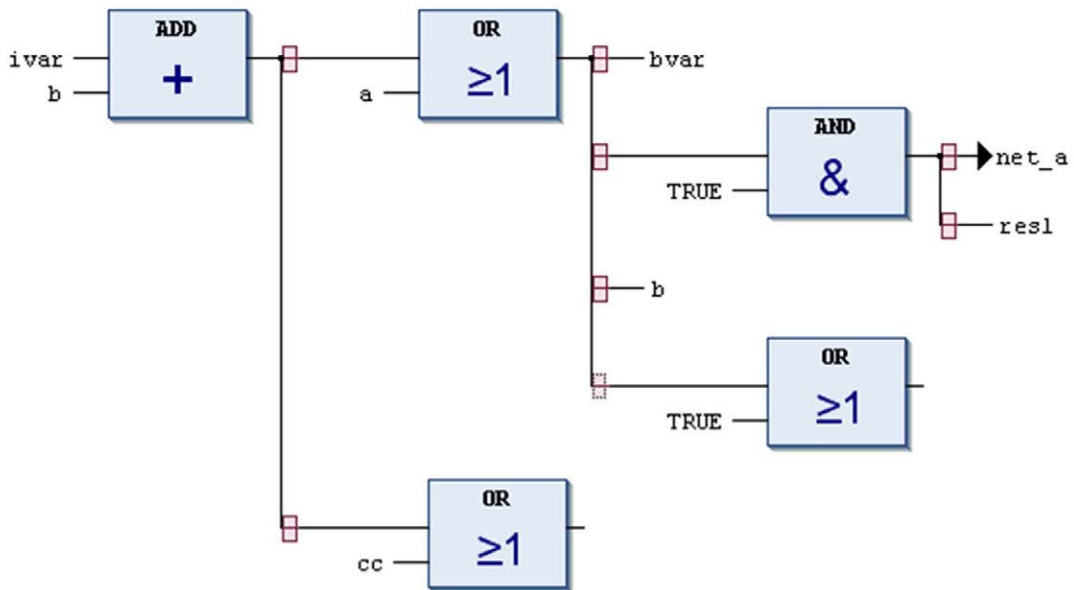
Branch / Hanging Coil in FBD/LD/IL

Overview

In a Function Block Diagram (*see page 260*) or Ladder Diagram (*see page 261*) 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 (*see page 278*)) in order to perform actions on this arm of the branch.

Branch markers

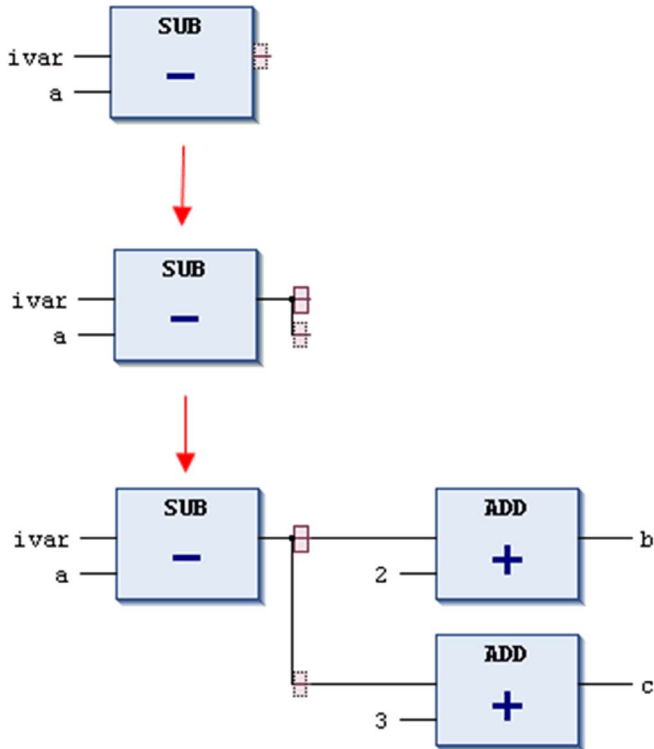


In FBD, insert a branch via command **Insert branch**. Alternatively, drag the element from the toolbox (*see page 290*). For the possible insert positions, refer to the description of the **Insert branch** command.

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.

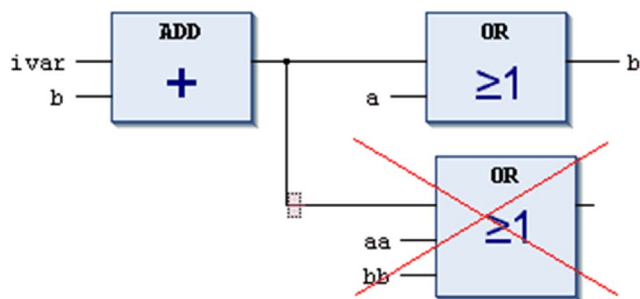
Network in FBD with inserted branch



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 DEL 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 ([see page 262](#)) 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 ([see page 299](#)) 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.

NOTE: The term branch is also used for another element that splits off a signal flow. This branch (*see page 296*) 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.

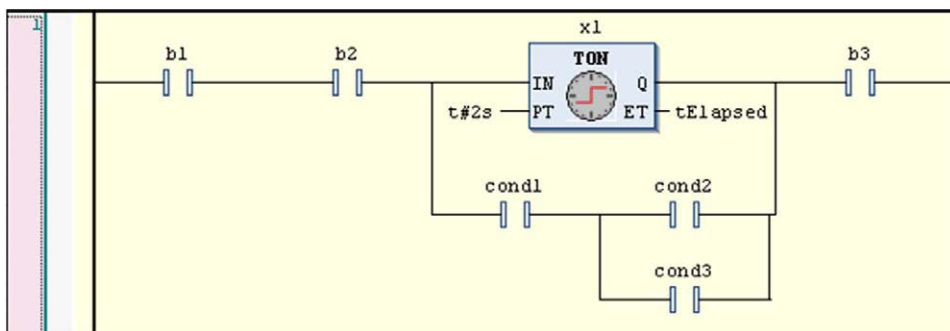
To insert a parallel branch with SCE function, select the function block box and execute the command **Insert Contact Parallel above** or **Insert Contact Parallel below**. This is only possible if the first input and the main output of the function block are of type BOOL.

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

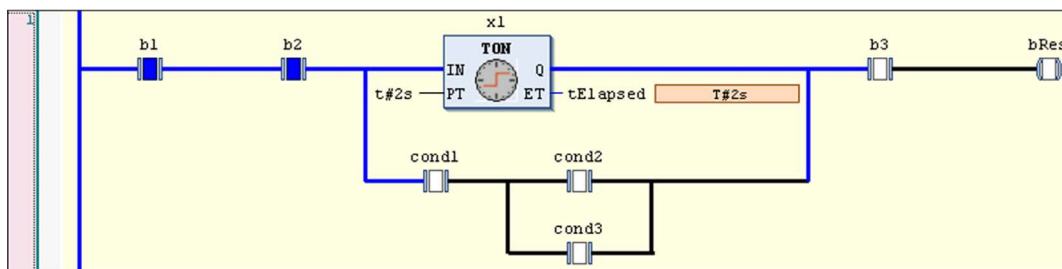


The processing is as shown in the following, whereby P_IN and P_OUT represent the boolean value at the input (split point) and output (junction point) of the parallel branch, respectively.

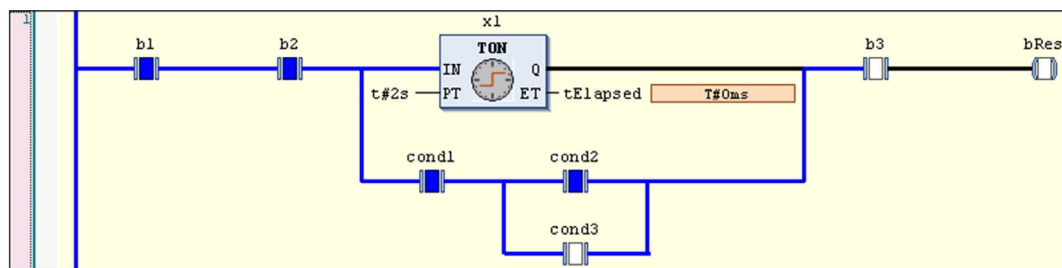
```
P_IN := b1 AND b2;
IF ((P_IN AND cond1) AND (cond2 OR cond3)) THEN
  P_OUT := P_IN;
ELSE
  x1(IN := P_IN, PT := {p 10}t#2s);
  tElapsed := x1.ET;
  P_OUT := x1.Q;
END_IF
bRes := P_OUT AND b3;
```

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:



Set/Reset in FBD/LD/IL

FBD and LD

A boolean output in FBD ([see page 260](#)) or correspondingly an LD ([see page 261](#)) coil can be set or reset. To change between the set states, use the respective command **Set/Reset** from the context 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.

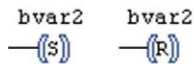
Set output in FBD



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 ([see page 302](#)).

IL

In an Instruction List, use the S and R ([see page 264](#)) operators to set or reset an operand.

Set/Reset Coil

Overview

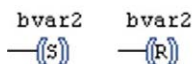
Coils (*see page 305*) can also be defined as set or reset coils.

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**.

Set coil, reset coil



Section 10.3

LD Elements

What Is in This Section?

This section contains the following topics:

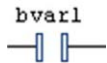
Topic	Page
Contact	304
Coil	305

Contact

Overview

This is an LD element.

In LD (*see page 261*) in its left part, each network contains 1 or several contacts. Contacts are represented by 2 vertical, parallel lines.

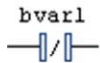


Contacts pass on condition ON (TRUE) or OFF (FALSE) from left to right. A boolean variable is assigned to each contact. If this variable is TRUE, the condition is passed from left to right and finally to a coil in the right part of the network. Otherwise the right connection receives the value FALSE.

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** (*see page 290*) or from another position within the editor (drag and drop).

FBD and IL

If you are working in FBD (*see page 260*) or IL (*see page 262*) 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.

bvar2
—()

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.

bvar2
—(/)

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* ([see page 302](#)).

FBD and IL

If you are working in FBD ([see page 260](#)) or IL ([see page 262](#)) 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.

Chapter 11

Continuous Function Chart (CFC) Editor

What Is in This Chapter?

This chapter contains the following topics:

Topic	Page
Continuous Function Chart (CFC) Language	308
CFC Editor	309
Cursor Positions in CFC	311
CFC Elements / ToolBox	313
Working in the CFC Editor	319
CFC Editor in Online Mode	322
CFC Editor Page-Oriented	324

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 ([see page 260](#)). 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 SoMachine, see the description of the CFC editor ([see page 309](#)).

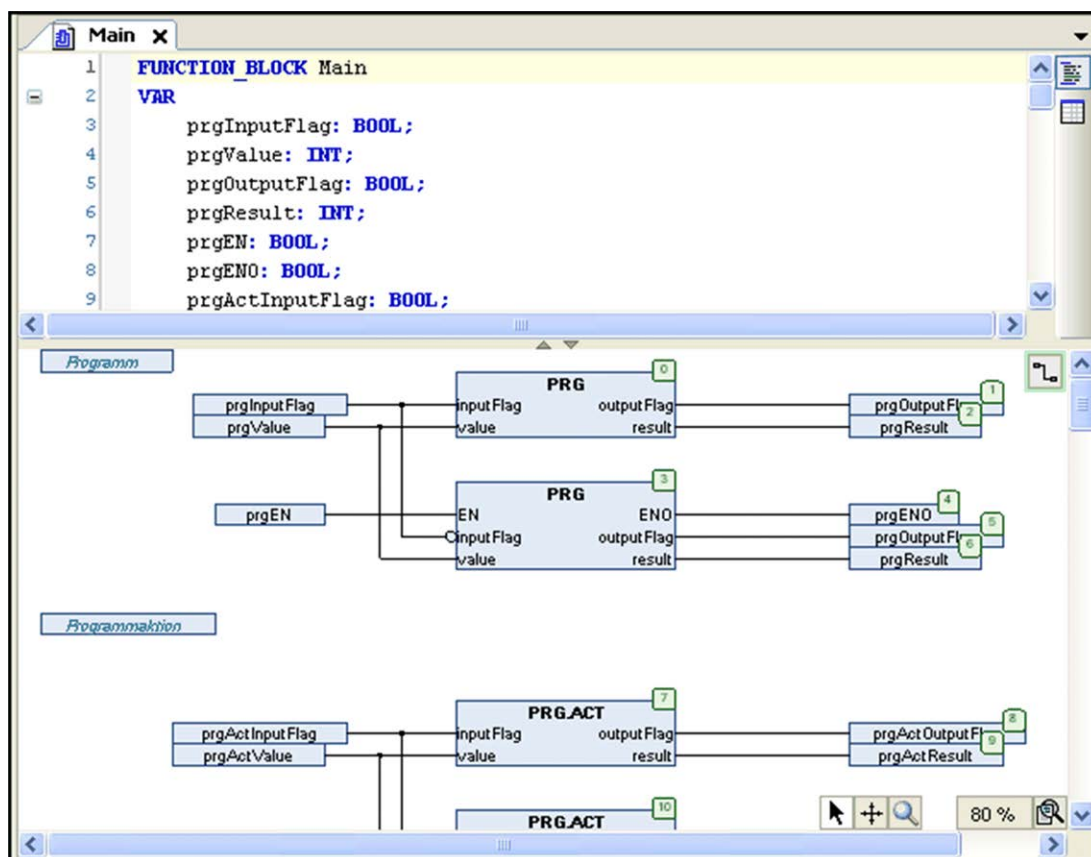
CFC Editor

Overview

The CFC editor is a graphical editor available for programming objects in the continuous function chart (CFC) programming language (*see page 308*), 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 (*see page 324*).

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 (*see page 382*) in its upper part.

CFC editor



The CFC editor in contrast to the FBD / LD editor allows free positioning (*see 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 (*see page 313*) 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 (*see page 319*). For complex charts, you can use connection marks (*see page 313*) 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 (*see page 313*) 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 context 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.

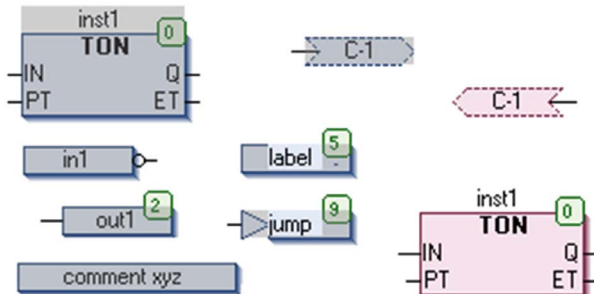
Possible cursor positions and an example of selected text:



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.

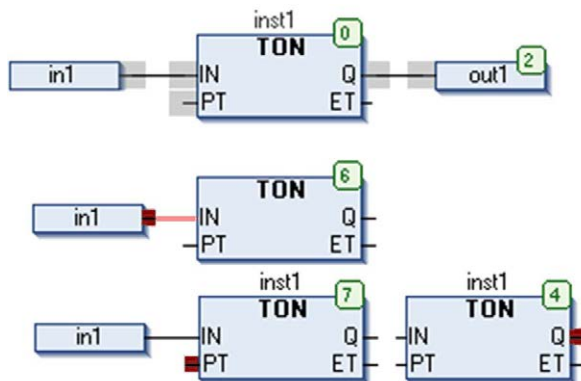
Possible cursor positions and example of a selected body:



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.

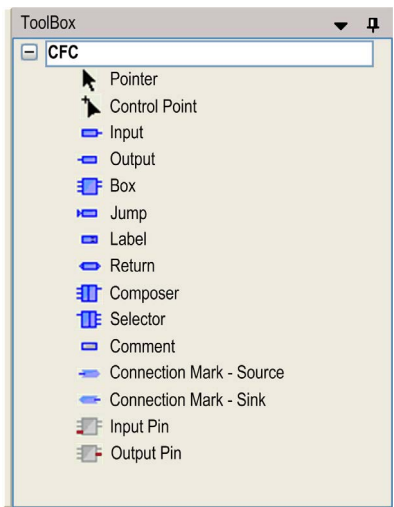
Possible cursor positions (gray shadows) and examples of selected output and input positions (red squares):



CFC Elements / ToolBox

Overview

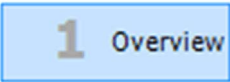
The graphical elements available for programming in the CFC editor (*see page 309*) window are provided by a toolbox. Open the toolbox in a view window by executing the command **ToolBox** in the **View** menu.



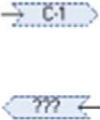
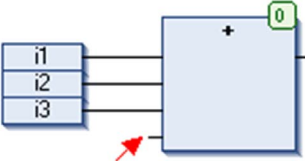
Select the desired element in the toolbox and insert (*see page 319*) it in the editor window via drag and drop.

Besides the programming elements, there is an entry  **Pointer**, 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.

Name	Symbol	Description
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 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.

Name	Symbol	Description
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.
connection mark – source connection mark – sink		You can use connection marks instead of a connection line (see 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 <i>Connection Mark</i>.
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.
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.

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
```

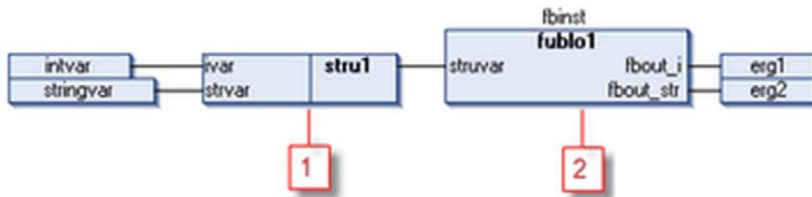
Declaration and implementation of function block `fublo1`:

```
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** (*see page 313*) 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** (*see page 313*) 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 (*see 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.

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 POU's and thus some existing assignments could be removed.

Moving

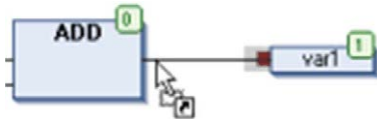
To move an element, select the element by clicking the element body (see possible cursor positions (*see page 311*)) and drag it, while keeping the mouse-button pressed, to the desired position. Then release the mouse-button to place the element. You also can use the **Cut** and **Paste** commands for 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* ([see page 311](#))), 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 will be created taking into account the other elements and connections. If the route of connection lines overlaps other connection lines, they are colored light-gray.

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 ([see page 313](#)).

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 context menu or press the DEL 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 context 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* ([see 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 SoMachine, Menu Commands, Online Help*). In case of page-oriented CFC, you can use the **Edit Page Size** command (*see SoMachine, Menu Commands, Online Help*).

CFC Editor in Online Mode

Overview

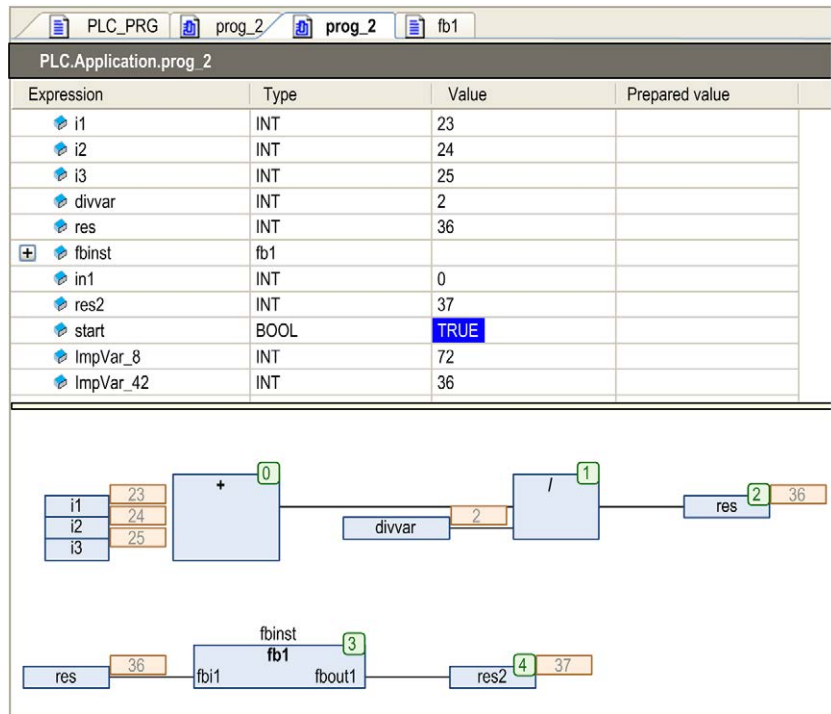
In online mode, the CFC editor provides views for monitoring. The views for writing and forcing the variables and expressions on the controller are described in separate chapters. The debugging functionality (breakpoints, stepping, and so on) is available as described below.

- For information on how to open objects in online mode refer to the description of the user interface in online mode ([see page 51](#)).
- 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 ([see page 387](#)).

Monitoring

The actual values are displayed in small monitoring windows behind each variable (inline monitoring).

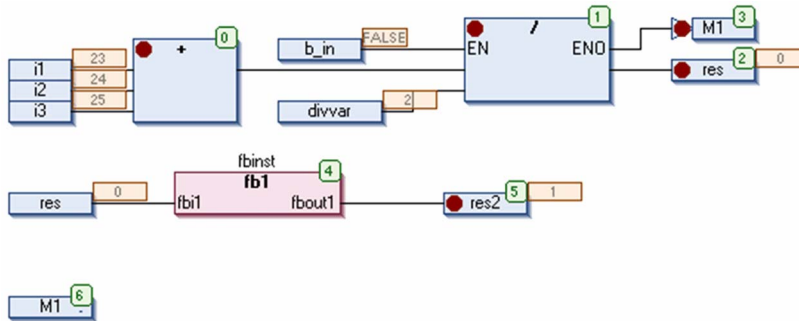
Online view of a program object PLC_PRG:



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.

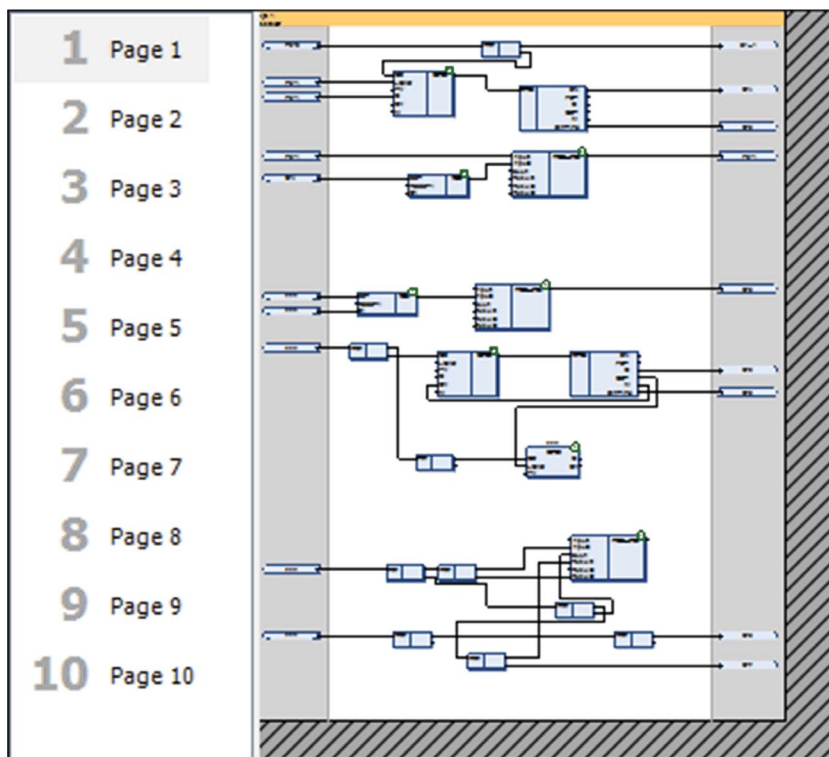
CFC Editor Page-Oriented

Overview

In addition to the CFC standard editor, SoMachine provides the CFC editor pagination. Besides the tools (*see page 313*) 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 POU's 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

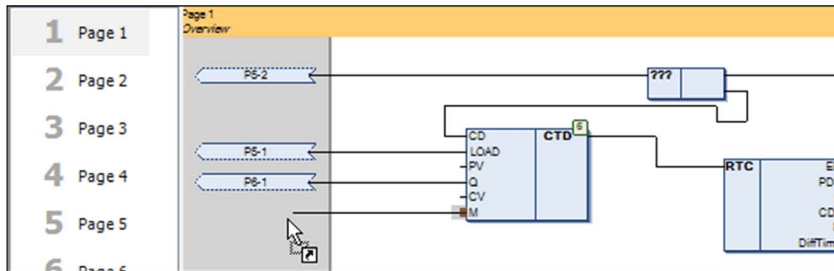


To change the size of the page execute the command **Edit Page Size**.

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.

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 ([see page 321](#))). You can change the execution order of elements only within the associated page. You cannot change the execution order of elements on different pages.

Chapter 12

Sequential Function Chart (SFC) Editor

What Is in This Chapter?

This chapter contains the following topics:

Topic	Page
SFC Editor	328
SFC - Sequential Function Chart Language	330
Cursor Positions in SFC	331
Working in the SFC Editor	333
SFC Element Properties	335
SFC Elements / ToolBox	337
Qualifier for Actions in SFC	349
Implicit Variables - SFC Flags	350
Sequence of Processing in SFC	355
SFC Editor in Online Mode	357

SFC Editor

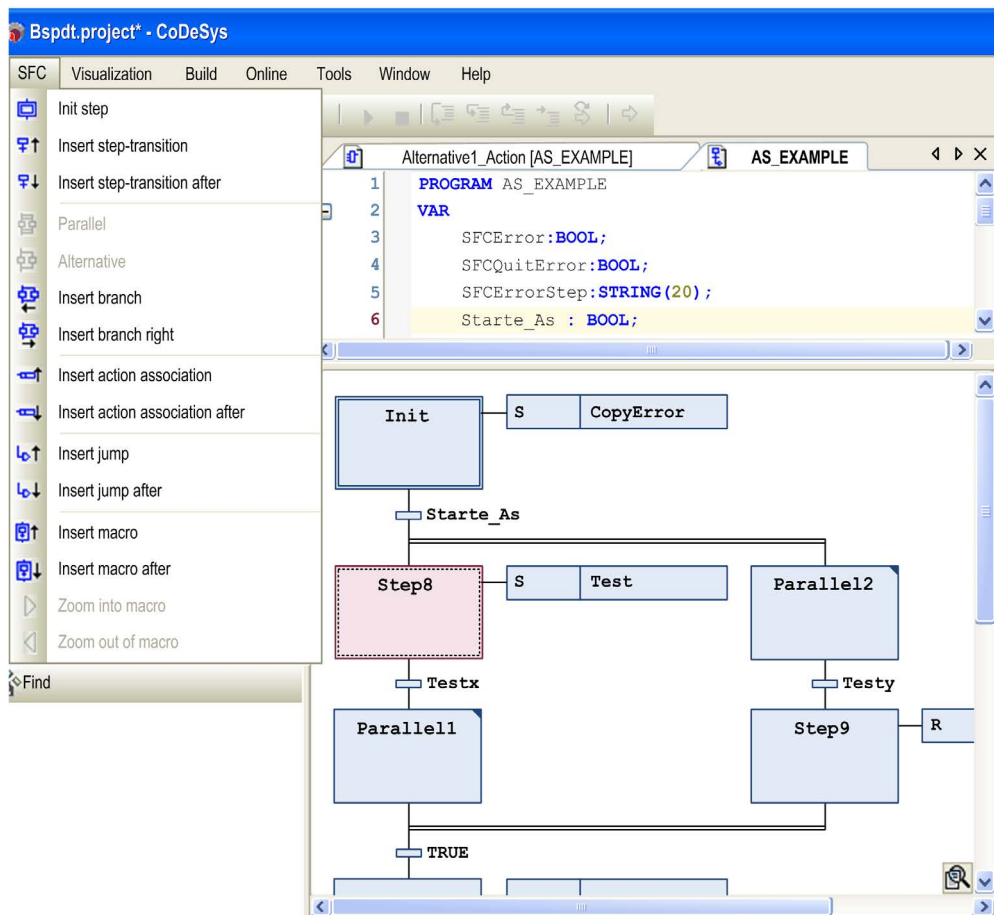
Overview

The SFC editor is available for programming objects in the IEC 61131-3 programming language SFC - Sequential Function Chart ([see page 330](#)). 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** ([see page 382](#)) in the upper part.

SFC editor



Working with the SFC Editor

The elements ([see page 337](#)) 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* ([see page 333](#)).

You can edit the properties of steps in a separate properties ([see page 335](#)) window. Among others, you can define the minimum and maximum time of activity for each step.

You can access implicit variables ([see page 350](#)) for controlling the processing of an SFC (for example, step status, timeout analyzation, reset).

SFC - Sequential Function Chart Language

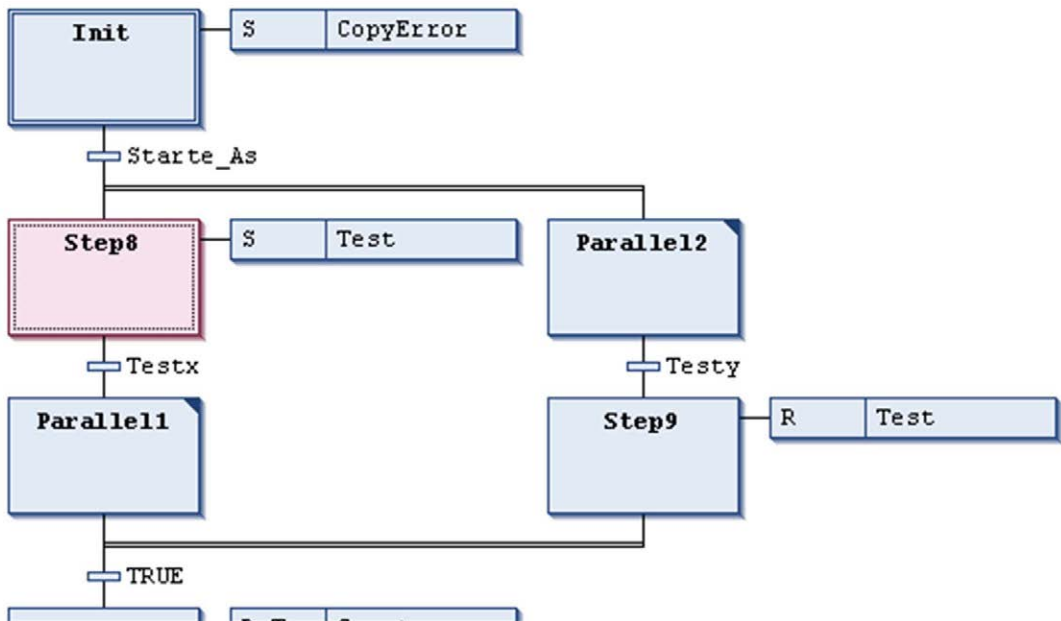
Overview

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* (see page 355).

For information on how to use the SFC editor in SoMachine, refer to the description of the *SFC Editor* (see page 328).

Example

Example for a sequence of steps in an SFC module:



Cursor Positions in SFC

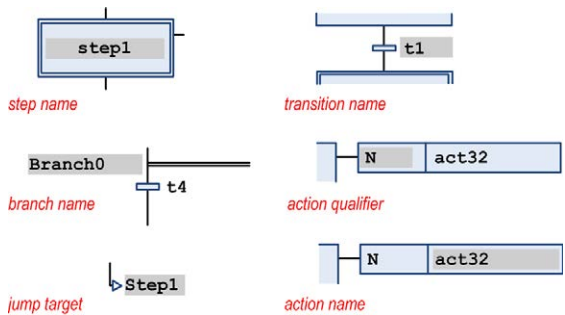
Overview

Possible cursor positions in an SFC diagram in the SFC editor ([see page 328](#)) 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:



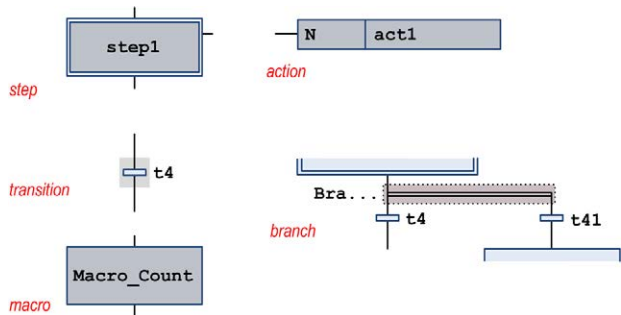
When you click a text cursor position, the string will become editable.

Select action name for editing:



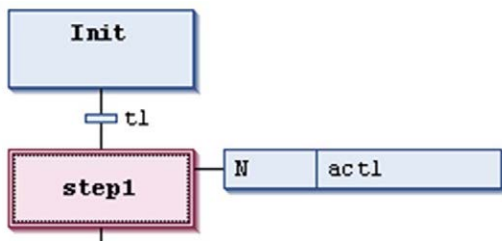
Cursor Positions in Element Bodies

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* ([see 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* ([see page 331](#)).

Navigating

Use the arrow keys to jump to the next or previous element in the chart.

Inserting Elements

To insert the particular SFC elements ([see page 337](#)), execute the respective commands from the **SFC** menu. For further information, refer to the description of the SFC editor commands (*see SoMachine, 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* ([see page 331](#)).

NOTE: In contrast to previous versions of SoMachine, 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.

Deleting Elements

Select the elements and execute the command **Delete** or press the DEL 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. The properties are grouped. You can open and close the particular group sections by using the plus or minus signs.

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. Double-click the value field, or select the value field and press the SPACE key in order to open a selection list from which you can choose one of the following access options: 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 in 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) (<i>see page 337</i>). 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.
Times:		Defines the minimum and maximum processing times for the step. NOTE: Time outs in steps are indicate by the implicit variable (<i>see page 350</i>) SFCErrror flag.
	Minimal active	Minimum length of time the processing of this step should take. 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. Permissible values: time according to IEC syntax (for example, t#8s) or TIME variable; default: t#0s
Actions:		Defines the actions (<i>see page 341</i>) to be performed when the step is active. Refer to the description of the Sequence of Processing in SFC (<i>see page 355</i>) 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 appropriate implicit variables to determine the status of actions and time outs via SFC flags (*see page 350*).

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* ([see page 333](#))

The following elements are available and are described in this chapter:

- step ([see page 337](#))
- transition ([see page 337](#))
- action ([see page 341](#))
- branch (alternative) ([see page 345](#))
- branch (parallel) ([see page 345](#))
- jump ([see page 347](#))
- macro ([see page 347](#))

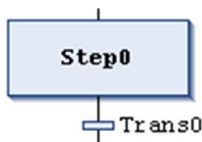
Step / Transition

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.

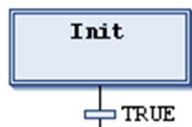
A step is represented by a box primarily containing an automatically generated step name. It is connected to the preceding and subsequent transition by a line. The box frame of the first step within an SFC, the initial step, is double-lined.

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 (see 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 (see page 341).

Recommendations on 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.

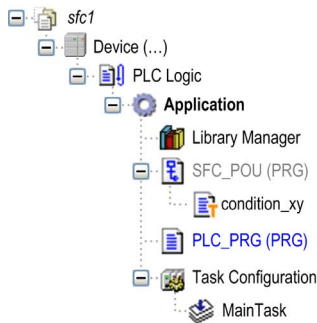
A transition condition can be defined in the following 2 ways:

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

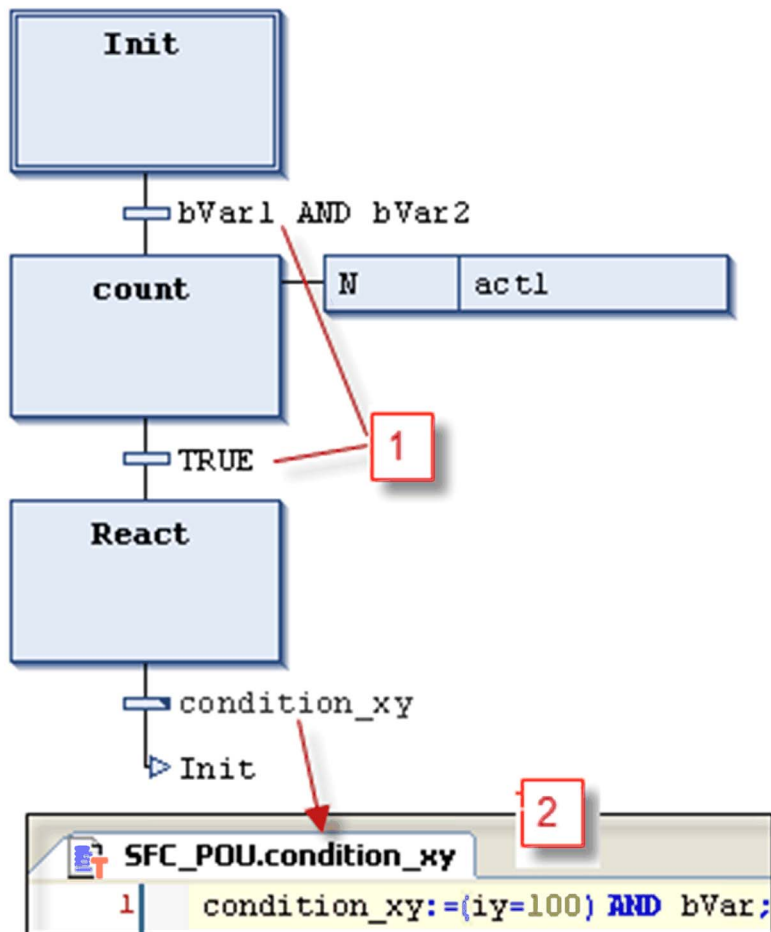
NOTE: If a transition produces multiple statements, assign the desired expression to a transition variable.

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):

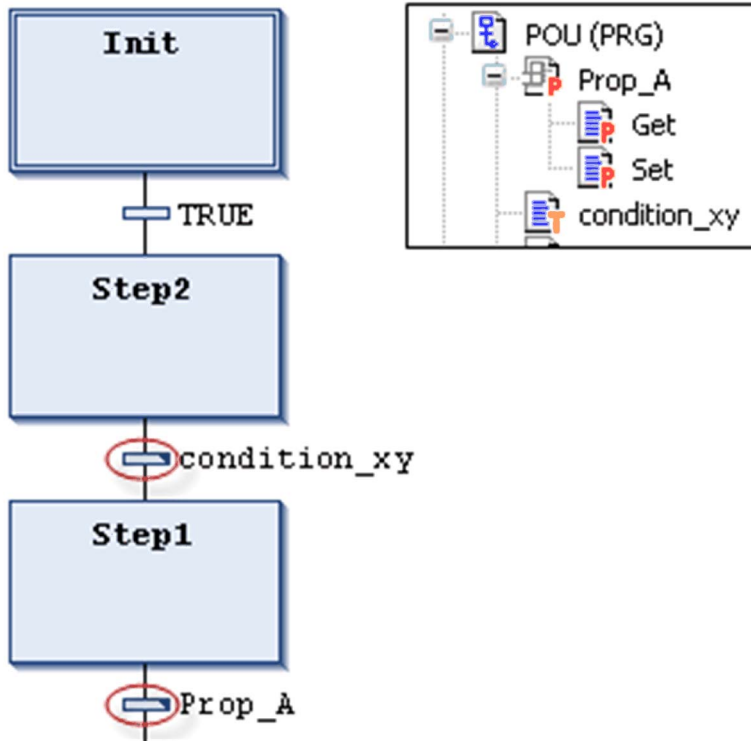


Examples of transitions:



- 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 SoMachine, 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>;`

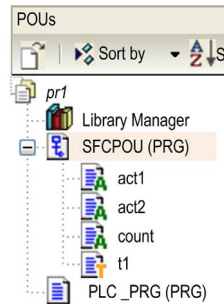
Example: `a=100;`

See also an example (`condition_xy`) in the figure *Examples of transitions*.

Action

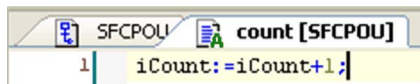
An action can contain a series of instructions written in one of the valid programming languages. It is assigned to a step and, in online mode, it will be processed according to the defined sequence of processing (*see page 355*).

Each action to be used in SFC steps must be available as a valid POU within the SFC POU or the project ().



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 to. Otherwise, an error will be detected during the build process.

Example of an action written in ST



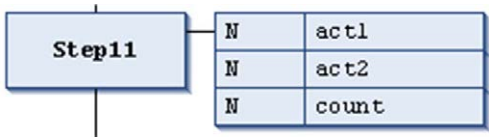
The IEC conforming and the IEC extending step actions are described in the following paragraphs.

IEC Conforming Step Action (IEC Action)

This is an action according to the IEC61131-3 standard which will be processed according to its qualifier (*see page 349*) 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.

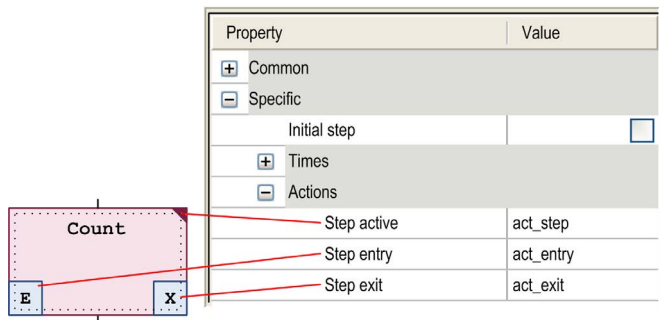
IEC Extending Step Actions

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:

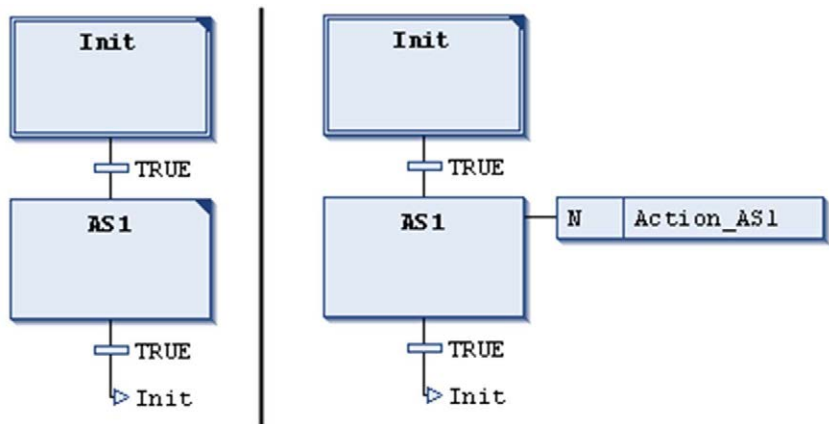
Action Type	Processing	Association	Representation
step entry action (step activated)	This type of step action will be processed as soon as the step has become active and before the step active action.	The action is associated to a step via an entry in the Step entry field of the step properties (see 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 (see 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 (see page 335).	It is represented by an X in the lower right corner of the respective step box.

IEC extending step actions



Example: Difference 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.



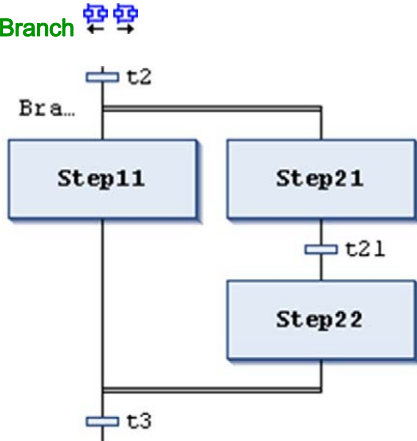
Action Action_AS1 is associated to step AS1 as a step action (left), or as an IEC action with qualifier N (right). Due to the fact that in both cases 2 transitions are used, it will take 2 controller cycles each before the initial step is reached again, assuming that a variable iCounter is incremented 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 ([see page 349](#)).

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 ([see page 345](#)) will be processed parallel (simultaneously). In the case of alternative branches ([see page 345](#)), 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 ([see page 347](#)).

Parallel Branch



A parallel branch has to begin and end with a step. Parallel branches can contain alternative branches or other parallel branches.

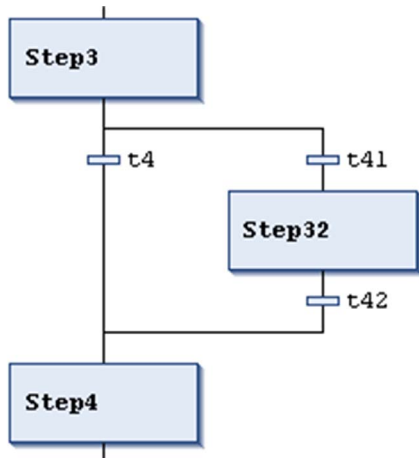
The horizontal lines before and after the branched area are double-lines.

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 ([see page 347](#)).

Alternative Branch 

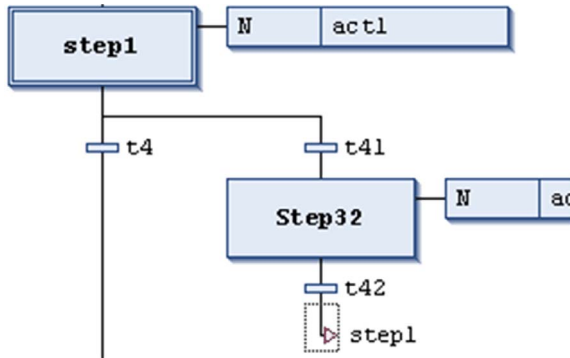
The horizontal lines before and after the branched area are simple lines.

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  

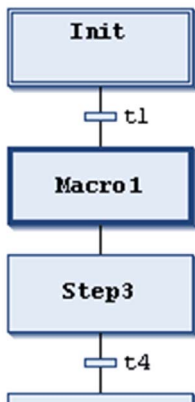
A jump is represented by a vertical connection line plus a horizontal arrow and the name of the jump target. It defines the next step to be processed as soon as the preceding transition is TRUE. You can use jumps to avoid that processing lines cross or lead upward.

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**.

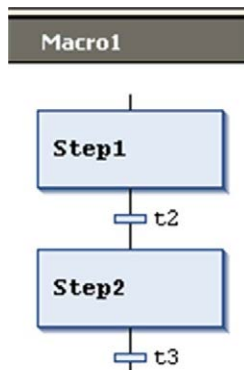
The target of the jump is specified by the associated text string which can be edited. It can be a step name or the label of a parallel branch.

Macro  

Main SFC editor view



Macro editor view for Macro01



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 ([see page 341](#)) 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.
R0	overriding reset	The action becomes deactivated.
S0	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 and will be executed once.
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.

Implicit Variables - SFC Flags

Overview

Each SFC step and IEC action provides implicitly generated variables for watching the status (see page 350) 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 (see 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 lecSFC.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: In contrast to previous versions of SoMachine, implicit variables for actions are preceded by an underscore in V4.0 and later.

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. This means that <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 (*see 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.

 **WARNING**

UNINTENDED EQUIPMENT OPERATION

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.

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 (*see 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 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.

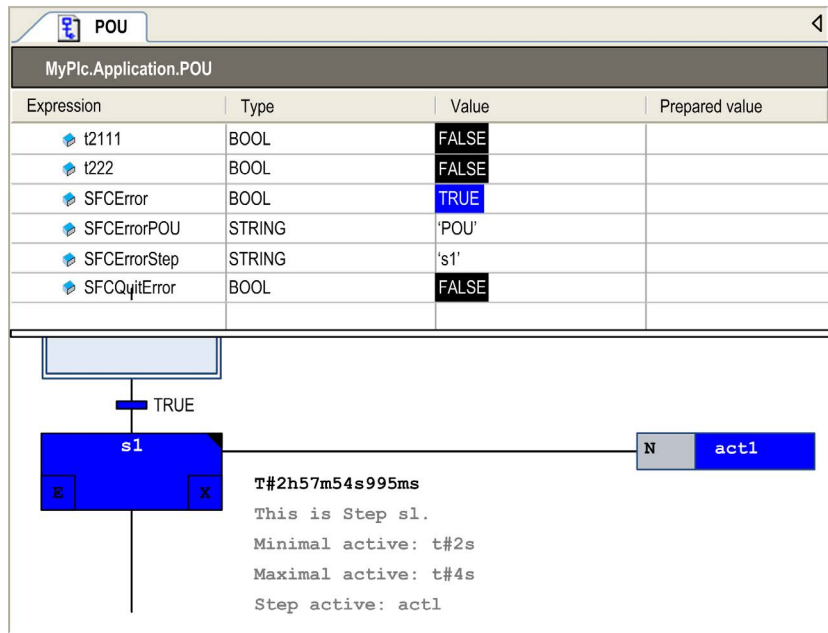
The following implicit variables (flags) can be used. For this purpose, you have to declare and activate them in the **SFC Settings** dialog box.

Variable	Type	Description
SFCInit	BOOL	If this variable becomes TRUE, the sequential function chart will be set back to the Init step (<i>see page 337</i>). 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).
SFCEnableLimit	BOOL	You can use this variable for the explicit activation (TRUE) and deactivation (FALSE) of the time control in steps via SFCError. This means, that 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. Precondition: SFCError must be defined.
SFCErrorPOU	STRING	This variable stores the name of the SFC POU in which a timeout has occurred. Precondition: SFCError must be defined.

Variable	Type	Description
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.
SFCTipSFCTipMode	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.

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 [\(see page 351\)](#)).

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
```

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 (<i>see page 341</i>) 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 (see page 341) 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 (see 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).
5. IEC actions	IEC actions (see page 341) 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 (see 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. Then 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 ([see page 350](#)) for determining the status of steps and actions or the execution of the chart.

SFC Editor in Online Mode

Overview

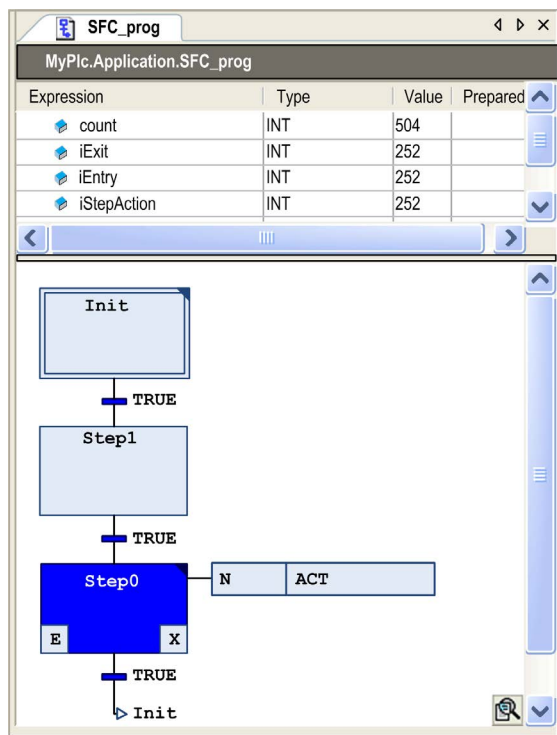
In online mode, the SFC editor provides views for monitoring (see below) and for writing and forcing the variables and expressions on the controller. Debugging functionality like for the other IEC languages (breakpoints, stepping, and so on) is not available in SFC. However, consider the following hints for debugging SFC:

- For information on how to open objects in online mode, refer to the description of the user interface in online mode ([see page 51](#)).
- 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* ([see page 387](#)). If you have declared implicit variables (SFC flags) ([see page 350](#)) 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 ([see page 355](#)) 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 ([see page 350](#)) 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



Chapter 13

Structured Text (ST) Editor

What Is in This Chapter?

This chapter contains the following sections:

Section	Topic	Page
13.1	Information on the ST Editor	360
13.2	Structured Text ST / Extended Structured Text (ExST) Language	366

Section 13.1

Information on the ST Editor

What Is in This Section?

This section contains the following topics:

Topic	Page
ST Editor	361
ST Editor in Online Mode	362

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.

Further Information

To select blocks, press 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 (*see page 382*) 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).

ST Editor in Online Mode

Overview

In online mode, the structured text editor (ST editor) provides views for monitoring ([see page 362](#)), 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 ([see page 364](#)).

- For information on how to open objects in online mode, refer to the description of the user interface in online mode ([see page 51](#)).
- For information on how to enter prepared values for variables in online mode, see *Forcing of variables* ([see page 362](#)).
- 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 ([see page 387](#)).

Monitoring

If the monitoring is not explicitly deactivated in the **Options** 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:

The screenshot displays the ST Editor interface for a program object named PLC_PRG. The top section shows a table of variable declarations with columns for Expression, Type, Value, Prepared value, Address, and Comment. Below the table, a code snippet is visible, showing the implementation of the variables.

Expression	Type	Value	Prepared value	Address	Comment
iVar	INT	2411		%MW9	
bVar	BOOL	TRUE		%QX0.3	
myStruct	TestStruct				Check address defined in "TestStru
nTest1	INT	2411		%MW0	
nTest2	INT	0		%MW2	
aVar	ARRAY [0..3] OF INT			%MW5	
fbinst	FB1				instance of function block FB1
fbinst.fbIn	INT	0			
fbinst.fbOut	INT	0			
fbinst.fbRet	INT	0			

```
1  iVar [2411] := iVar [2411] + 1; (* counter *)
2  bVar [TRUE] := TRUE;
3  myStruct.nTest1 [2411] := iVar [2411];
4  aVar [2] [2411] := iVar [2411];
5  erg [0] := fbinst.fbOut [0]; RETURN
```

Forcing of Variables

In addition to the possibility to enter 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 rising dialog box.

Dialog box **Prepare Value**

Prepare Value

Expression:

Type:

Current value:

What do you want to do?

☒ Prepare a new value for the next write or force operation:

☐ Remove preparation with a value.

☐ Release the force, without modifying the value.

☐ Release the force and restore the variable to the value it had before forcing it.

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 (*see page 368*) 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:

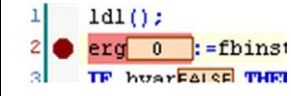
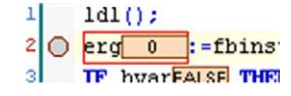
At the beginning of the line.

```
{{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.

Section 13.2

Structured Text ST / Extended Structured Text (ExST) Language

What Is in This Section?

This section contains the following topics:

Topic	Page
Structured Text ST / Extended Structured Text ExST	367
Expressions	368
Instructions	370

Structured Text ST / Extended Structured Text ExST

Overview

Structured Text is a textual high-level programming language, similar to PASCAL or C. The program code is composed of expressions (*see page 368*) and instructions (*see page 370*). 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;
```

Extended Structured Text (ExST) is a SoMachine-specific extension to the IEC 61131-3 standard for Structured Text (ST). Examples: assignment as expression, set/reset operators

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 (*see page 639*), operands (*see page 731*), and/or assignments. An operand can be a constant, a variable, a function call, or another expression.

Examples

33	(* Constant *)
ivar	(* Variable *)
fct(a, b, c)	(* Function call*)
a AND b	(* Expression *)
(x*y) / z	(* Expression *)
real_var2 := int_var;	(* Assignment, see below *)

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	..

Operation	Symbol	Operating Level
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:

<code>int_var1 := int_var2 := int_var3 + 9;</code>	(* int_var1 and int_var2 both equal to the value of int_var3 + 9*)
<code>real_var1 := real_var2 := int_var;</code>	(* correct assignments, real_var1 and real_var2 will get the value of int_var *)
<code>int_var := real_var1 := int_var;</code>	(* a message will be displayed due to type mismatch real-int *)
<code>IF b := (i = 1) THEN i := i + 1; END_IF</code>	(*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 ([see page 368](#)).

The following instructions can be used in ST:

Instruction	Example
assignment (see page 371)	A:=B; CV := CV + 1; C:=SIN(X);
Calling a function block (see page 372) and using the function block output	CMD_TMR(IN := %IX5, PT := 300); A:=CMD_TMR.Q
RETURN (see page 372)	RETURN;
IF (see page 372)	D:=B*B; IF D<0.0 THEN C:=A; ELSIF D=0.0 THEN C:=B; ELSE C:=D; END_IF;
CASE (see page 373)	CASE INT1 OF 1: BOOL1 := TRUE; 2: BOOL2 := TRUE; ELSE BOOL1 := FALSE; BOOL2 := FALSE; END_CASE;
FOR (see page 374)	J:=101; FOR I:=1 TO 100 BY 2 DO IF ARR[I] = 70 THEN J:=I; 'EXIT; END_IF; END_FOR;
WHILE (see page 375)	J:=1; WHILE J<= 100 AND ARR[J] <> 70 DO J:=J+2; END_WHILE;
REPEAT (see page 376)	J:=-1; REPEAT J:=J+2; UNTIL J= 101 OR ARR[J] = 70 END_REPEAT;
EXIT (see page 377)	EXIT;

Instruction	Example
CONTINUE (<i>see page 377</i>)	CONTINUE;
JMP (<i>see page 377</i>)	<pre> label1: i:=i+1; IF i=10 THEN JMP label2; END_IF JMP label1; label2: </pre>
empty instruction	;

Assignment Operators

Standard 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 (*see page 652*) which has the same function.

Example

```
Var1 := Var2 * 10;
```

After completion of this line, Var1 has the tenfold value of Var2.

Set operator S=

The value will be set: if it is once set to TRUE, it will remain TRUE.

Example

```
a S= b;
```

a gets the value of b: if once set to TRUE, it will remain TRUE, even if b becomes FALSE again.

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 (*see page 368*). This is an extension to the IEC 61131-3 standard.

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 <Var 1> 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 RESULTING IN UNINTENDED EQUIPMENT OPERATION

Ensure that the variable type used in FOR instructions is of a sufficient capacity (has a great enough upper limit) 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 RESULTING IN UNINTENDED EQUIPMENT OPERATION

Ensure that the WHILE loop will be terminated within the instructions of the loop by creating a FALSE 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:

```
WHILE Counter<>0 DO
  Var1 := Var1*2;
  Counter := Counter-1;
END_WHILE
```

The REPEAT instruction has not yet been introduced so moving paragraph (with modification) to below.

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 RESULTING IN UNINTENDED EQUIPMENT OPERATION

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.

The WHILE and REPEAT loops are, in a certain sense, more powerful than the FOR loop since you do not have to know the number of cycles before executing the loop. In some cases, you will therefore only be able to work with these two loop types. If, however, the number of the loop cycles is clear, then a FOR loop is preferable since, in most cases, it inherently excludes endless loops (see hazard message in the FOR loop paragraph ([see page 374](#))).

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 RESULTING IN UNINTENDED EQUIPMENT OPERATION

Ensure that the use of the JMP instruction is conditional such that it does not result in an infinite, or endless, loop.

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

There are 2 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.*)
- Single-line comments as an extension to the IEC 61131-3 standard: // denotes the start of a comment that ends with the end of the line. Example: // This is a comment.

You can place the comments everywhere within the declaration or implementation part of the ST editor.

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.

Part V

Object Editors

What Is in This Part?

This part contains the following chapters:

Chapter	Chapter Name	Page
14	Declaration Editors	381
15	Device Type Manager (DTM) Editor	389
16	Data Unit Type (DUT) Editor	391
17	Global Variables List (GVL) Editor	393
18	Network Variables List (NVL) Editor	395
19	Task Editor	419
20	Watch List Editor	429
21	Tools Within Logic Editors	435

Chapter 14

Declaration Editors

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 1 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 switch 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. This means, 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 (*see page 387*) 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 (*see page 519*).

What Is in This Chapter?

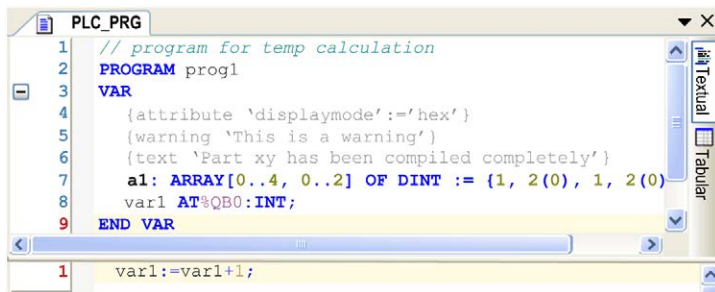
This chapter contains the following topics:

Topic	Page
Textual Declaration Editor	382
Tabular Declaration Editor	383
Declaration Editor in Online Mode	387

Textual Declaration Editor

Overview

Textual editor view

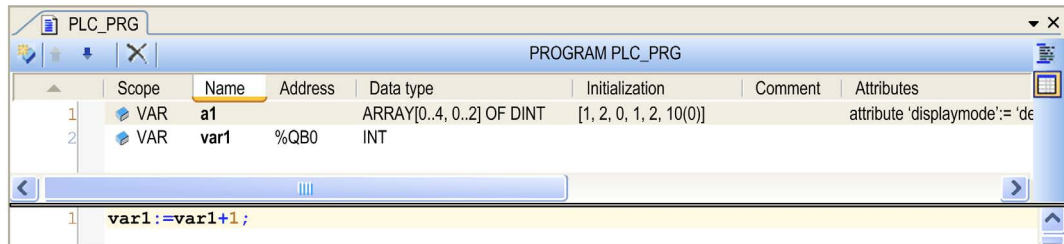


Behavior and appearance are determined by the respective current text editor settings in the **Options** and **Customize** dialog boxes. There you can define the default settings for highlight coloring, line numbers, tabs, indenting, and many more options. The usual editing functions are available such as copy and paste. Block selection is possible by pressing the ALT key while selecting the desired text area with the mouse.

Tabular Declaration Editor

Overview

Tabular editor view



The tabular view of the editor provides columns for the usual definitions for variable declaration (*see page 519*): **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 context 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. This is useful especially in case of 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 context menu.

You can sort the list of declarations according to each of the columns by clicking the header of the respective column. The column which currently determines the order is indicated by an arrow symbol:

arrow up = ascending order

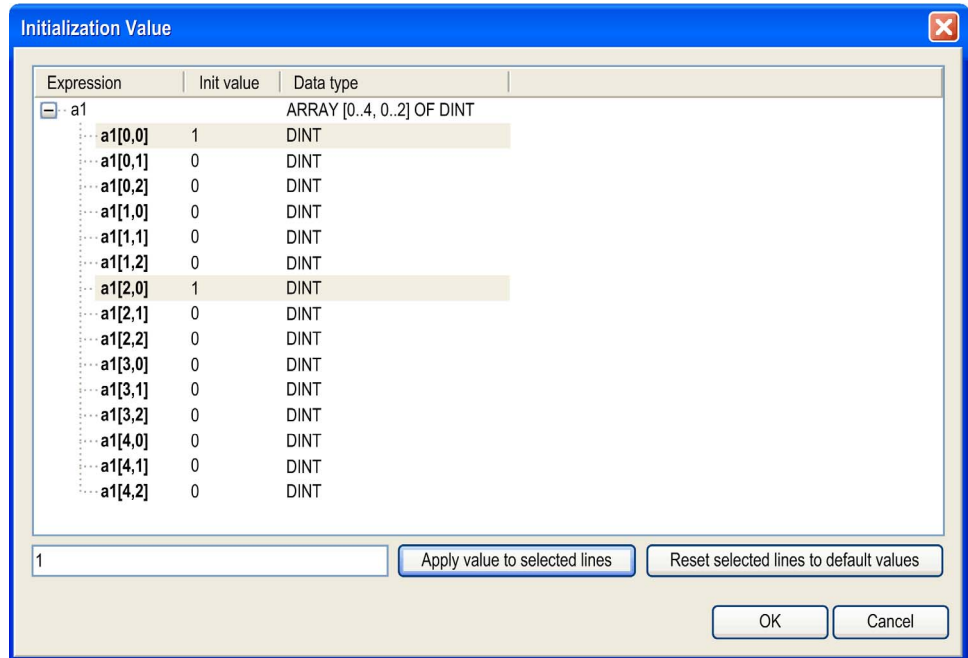
arrow down = descending order

Each further click in the column header changes between ascending and descending order.

To delete one or several declarations, select the respective lines and press the DEL key or execute the **Delete** command from the context menu or click the  button in the toolbar.

Initialization Value

Initialization value dialog box



The dialog box titled "Initialization Value" contains a table with three columns: "Expression", "Init value", and "Data type". The table lists various array elements of a variable named "a1". The "Init value" column shows the current initialization values, with some elements highlighted in yellow. Below the table is a text input field containing the number "1". To the right of the input field are two buttons: "Apply value to selected lines" and "Reset selected lines to default values". At the bottom right of the dialog are "OK" and "Cancel" buttons.

Expression	Init value	Data type
a1		ARRAY [0..4, 0..2] OF DINT
a1[0,0]	1	DINT
a1[0,1]	0	DINT
a1[0,2]	0	DINT
a1[1,0]	0	DINT
a1[1,1]	0	DINT
a1[1,2]	0	DINT
a1[2,0]	1	DINT
a1[2,1]	0	DINT
a1[2,2]	0	DINT
a1[3,0]	0	DINT
a1[3,1]	0	DINT
a1[3,2]	0	DINT
a1[4,0]	0	DINT
a1[4,1]	0	DINT
a1[4,2]	0	DINT

1

Apply value to selected lines Reset selected lines to default values

OK Cancel

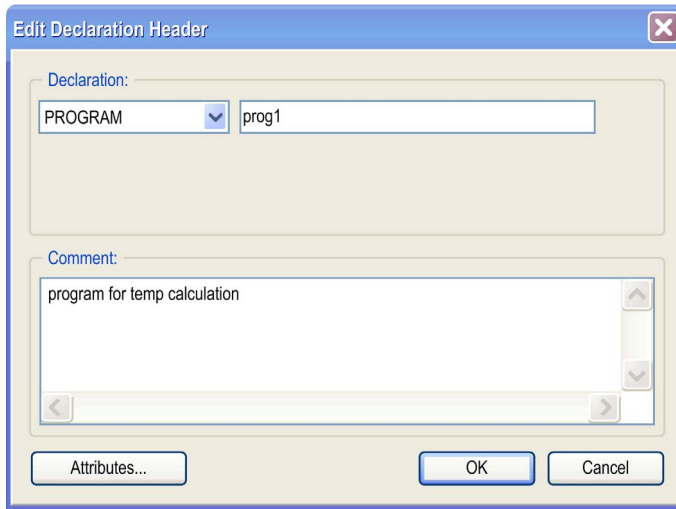
The **Expressions** of the variable are displayed with the current initialization values. Select the desired variables and edit the initialization value in the field below the listing. Then click the **Apply value to selected lines** button. To restore the default initializations, click the **Reset selected lines to default values** button.

Press CTRL + ENTER to insert line breaks in the **Comment** entry.

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



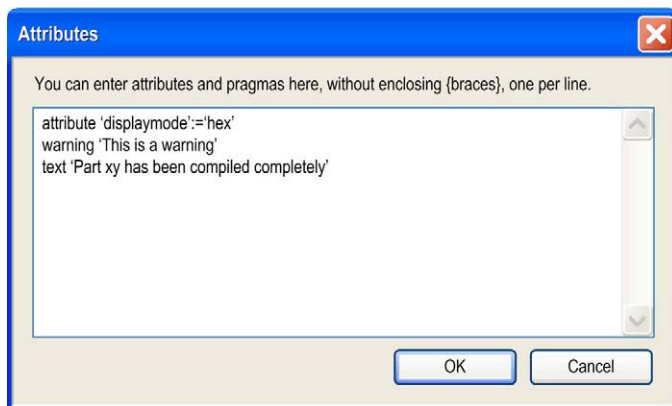
The **Edit Declaration Header** dialog box provides the following elements:

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 **Attributes** 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 ([see page 382](#)).

Attributes dialog box



Declaration Editor in Online Mode

Overview

After log-in to the controller, each object which has already been opened in a window in offline mode will automatically be displayed in online view.

The online view of the declaration editor presents a table similar to that used in watch views (*see page 433*). The header line shows the actual object path <device name>.<application name>. The table for each watch expression shows the type and current 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.

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 will open to select a value. In case of a boolean variable, the handling is even easier.

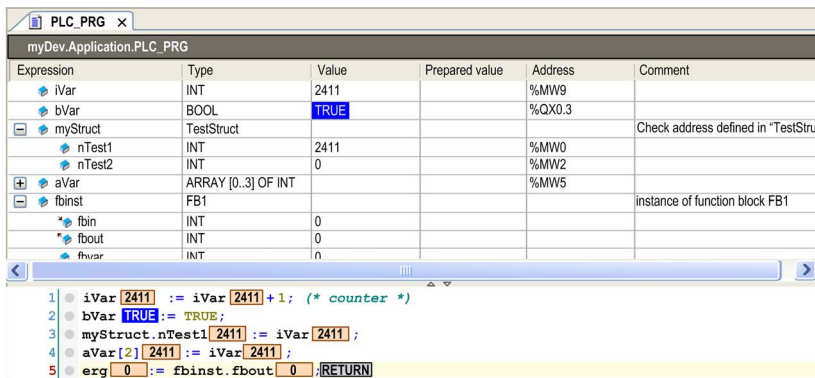
You can toggle boolean preparation values by use of the RETURN or SPACE key according to the following order:

- 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 will show 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



Expression	Type	Value	Prepared value	Address	Comment
iVar	INT	2411		%MW9	
bVar	BOOL	TRUE		%QX0.3	
myStruct	TestStruct				Check address defined in "TestStru
nTest1	INT	2411		%MW0	
nTest2	INT	0		%MW2	
aVar	ARRAY [0..3] OF INT			%MW5	
fbinst	FB1				instance of function block FB1
fbin	INT	0			
fbout	INT	0			
fivar	INT	n			

```

1 iVar[2411] := iVar[2411] + 1; (* counter *)
2 bVar[TRUE] := TRUE;
3 myStruct.nTest1[2411] := iVar[2411];
4 aVar[2][2411] := iVar[2411];
5 erg[0] := fbinst.fbout[0] - RETURN

```

Chapter 15

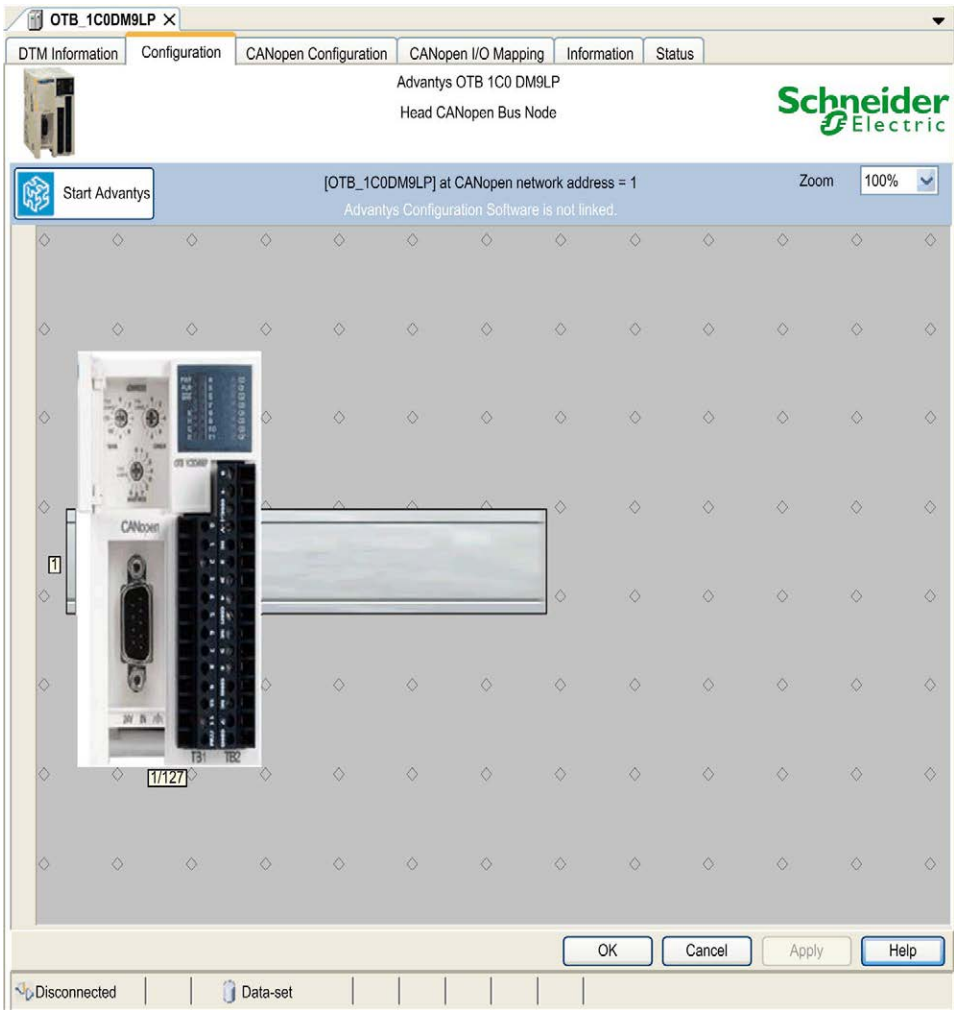
Device Type Manager (DTM) Editor

DTM Editor

Overview

The DTM editor view depends on the Device Type Manager.

The graphic provides an example of a DTM editor.



For further information on DTMs, refer to the *SoMachine Device Type Configuration (DTM) User Guide* (see *SoMachine, Device Type Manager (DTM), User Guide*).

For a list of the DTM versions currently supported by SoMachine, refer to the Release Notes of your SoMachine installation.

Chapter 16

Data Unit Type (DUT) Editor

Data Unit Type Editor

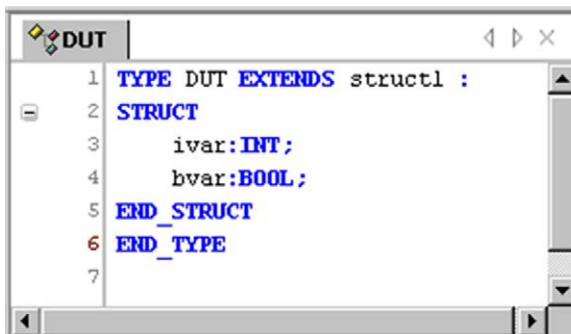
Overview

You can create user-defined data types (*see page 600*) 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



Chapter 17

Global Variables List (GVL) Editor

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 offline 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



Chapter 18

Network Variables List (NVL) Editor

What Is in This Chapter?

This chapter contains the following sections:

Section	Topic	Page
18.1	Information on the NVL Editor	396
18.2	General Information on Network Variables	397

Section 18.1

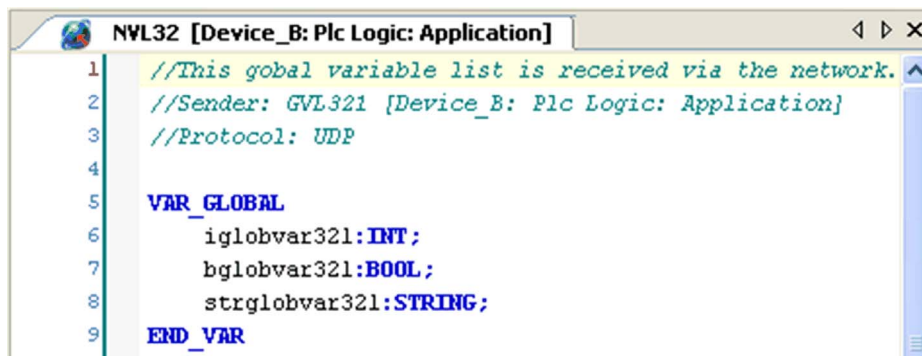
Information on the NVL Editor

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 offline 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 ([see page 517](#)) of global variables between them.

NVL editor

A screenshot of the NVL32 editor window. The title bar reads "NVL32 [Device_B: Plc Logic: Application]". The editor contains the following text:

```
1 //This gobal variable list is received via the network.  
2 //Sender: GVL321 [Device_B: Plc Logic: Application]  
3 //Protocol: UDP  
4  
5 VAR_GLOBAL  
6     iglobvar321:INT;  
7     bglobvar321:BOOL;  
8     strglobvar321:STRING;  
9 END_VAR
```

Section 18.2

General Information on Network Variables

What Is in This Section?

This section contains the following topics:

Topic	Page
Introduction to Network Variables List (NVL)	398
Configuring the Network Variables Exchange	401
Network Variables List (NVL) Rules	406
Operating State of the Sender and the Receiver	408
Example	409
Compatibility	415

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 of SoMachine. 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.

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

UNINTENDED MACHINE OPERATION DUE TO INCORRECT MACHINE COMMUNICATION

- You must consider the potential failure modes of control paths and, for certain critical control functions, provide a means to achieve a safe state during and after a path failure, including power outages and system restarts.
- Separate or redundant control paths must be provided for critical control functions.
- You must give consideration to the implications of unanticipated transmission delays or failures of the link.
- Observe all accident prevention regulations and local safety regulations.
- Each implementation of equipment using this feature must be individually and thoroughly tested for proper operation before being placed into service.

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

You can use Diagnostic (*see SoMachine, Network Variable Configuration, SE_NetVarUdp Library Guide*) and Error Management (*see SoMachine, 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 2 types of lists:

- Global Variables Lists (GVL) in a sending controller (sender)
- Global Network Variables List (GNVL) in a receiving controller (receiver)

The corresponding GVL and GNVL contain the same variable declarations. You can view their contents in the respective editor that opens after double-clicking the GVL or GNVL node in the **Devices** pane.

A GVL 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 all controllers that have a corresponding GNVL.

NOTE: For network variables exchange, the respective network libraries must be installed. This is done automatically for the standard network type UDP as soon as the network properties for a GVL are set.

Network variables are broadcasted from the GVL (sender) to one or more GNVL (receivers). For each controller you can define GVLs as well as GNVLs. Thus each controller can act as sender as well as receiver.

A sender GVL can be provided by the same or by another project. So, when creating a GNVL, the sender GVL can either be chosen from a selection list of all available GVLs 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 GVL.

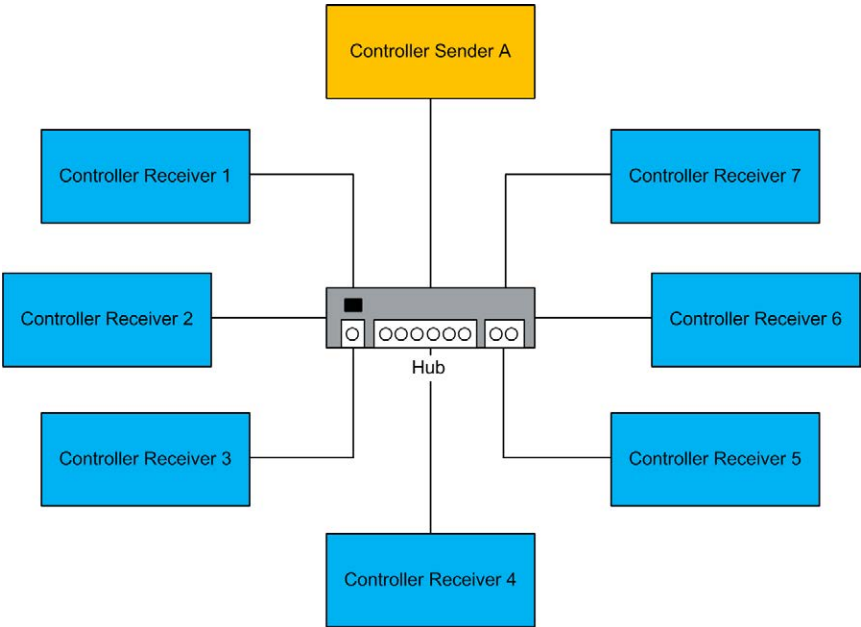
NOTE: An export file is needed if the sender GVL to be used is defined within another project.

NVL Considerations

The following table shows the list of controllers that support the network variables list (NVL) functionality:

Function Name	M238	M241	M251	M258 LMC058	XBTGC XBT GK XBT GT	ATV IMC
Network Variables List	No	Yes	Yes	Yes	Yes	Yes*
*ATV IMC supports NVL only in freewheeling tasks.						

The figure shows a network consisting of 1 sender and the recommended maximum of 7 receivers:



Controller Sender A: Sender with the global variables list (GVL) and receiver controller with global network variables lists (GNVLs)

Controller Receiver 1...7: Receivers (with GNVL) from A and sender controller (GVL) 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 SoMachine **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 global variables list (GVL) for the sender.
5	Define the global network variables list (GNVL) for the receiver.

An example with further information is provided in the Appendix (*see page 409*).

Global Variables List

To create the GVL for the sender, define the following network properties in the **GVL → Properties → Network properties** dialog box:

The screenshot shows a Windows-style dialog box titled "Properties - GVL [Device: PLC Logic: Application]". It has five tabs: "Common", "Link To File", "Access control", "Network properties" (which is selected), and "Build". In the "Network properties" tab, there are several configuration options: "Network type" is a dropdown menu set to "UDP" with a "Settings..." button next to it; "Task" is a dropdown menu set to "MAST"; "List identifier" is a text box containing the number "1". Below these are several checkboxes: "Pack variables" (checked), "Transmit checksum" (unchecked), "Acknowledgement" (unchecked), "Cyclic transmission" (checked), "Transmit on change" (unchecked), and "Transmit on event" (unchecked). To the right of the "Cyclic transmission" checkbox are three more text boxes: "Interval" set to "T#50ms", "Minimum gap" set to "T#20ms", and an empty "Variable" box. At the bottom of the dialog are three buttons: "OK", "Cancel", and "Apply".

Description of parameters

Parameter	Default Value	Description
Network type	UDP	Only the standard 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, we recommend to 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.
List identifier	1	Enter a unique number for each GVL on the network. It is used by the receivers for identifying the variables list (see page 407).
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.

Parameter	Default Value	Description
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 (<i>see SoMachine, Network Variable Configuration, SE_NetVarUdp Library Guide</i>).
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 (<i>see SoMachine, Network Variable Configuration, SE_NetVarUdp Library Guide</i>) 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.
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 GVL sender.
Broadcast Address	255.255.255.255	Enter a specific broadcast IP address for your application.

Global Network Variables List (GNVL)

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, a GNVL object can only be added to an application if a global variables list (GVL) 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 GNVL, 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 GNVL.
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 GVLs currently available in the project	Select the sender's GVL from the list of all sender GVLs with network properties currently available in the project. Select the entry Import from file from the list to use a GVL 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 standard Windows Explorer window that allows you to browse to the export file *.gv/you created from a GVL in another project. For further information refer to the <i>How to Add a GNVL From a Different Project</i> paragraph below.

How to Add a GNVL in the Same Project

When you add a GNVL via the **Add Object** dialog box, all appropriate GVLs that are found within the current project for the current network are provided for selection in the **Sender** list box. GVLs from other projects must be imported (see the *How to Add a GNVL From a Different Project* paragraph below).

Due to this selection, each GNVL in the current controller (sender) is linked to 1 specific GVL 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 GNVL.

How to Add a GNVL From a Different Project

Alternatively to directly choosing a sender GVL from another controller, you can also specify a GVL export file you had generated previously from the GVL by using the **Link to file** properties. This allows you to use a GVL 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.

GNVL Properties

If you double-click a **GNVL** item in the **Devices** tree, its content will be displayed on the right-hand side in an editor. But the content of the GNVL cannot be edited, because it is only a reference to the content of the corresponding GVL. The exact name and the path of the sender that contains the corresponding GVL is indicated at the top of the editor pane together with the type of network protocol used. If the corresponding GVL is changed, the content of the GNVL is updated accordingly.

Network Variables List (NVL) Rules

Rules on the Amount of Data

Because of some performance limitations, respect the following rules:

Number	Rule
1	Data transmission from one GVL (sender) to one GNVL (receiver) should not exceed 200 bytes.
2	Data exchange between several GVLs (senders) of one controller and their associated GNVLS (receivers) should not exceed 1000 bytes of variables.

Rules on the Number of Datagrams

To limit the maximum cycle time of NVL tasks, respect the following recommendations:

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. A notification Received overflow is raised in the diagnostics data (see <i>SoMachine, Network Variable Configuration, SE_NetVarUdp Library Guide</i>). One datagram can contain up to 256 bytes. That means that you should not exceed the limit of 5120 bytes of data received by 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 <i>SoMachine, Network Variable Configuration, SE_NetVarUdp Library Guide</i>). One datagram can contain up to 256 bytes. That means that you should not exceed the limit of 5120 bytes of data transmitted on one sender controller.

If the number of received / transmitted datagrams per cycle exceeds the limit several times, the following may happen:

- loss of UDP (user datagram protocol) 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 GVLs (Senders)

Define a maximum of 7 GVLs per controller (sender) to help maintain performance transparency.

Rules on Task Cycle Times of GVLs (Senders) and GNVLs (Receivers)

To help avoid reception overflow, you must define a cycle time for the task that manages the GVL transmission that is at least two times greater than the cycle time of the task that manages the GNVL reception.

Rules on the List Identifier Protection

The NVL function includes a list identifier checking:

The list identifier helps to avoid that a GVL (sender) from two separate controllers with the same list identifier (see dialog box **GVL → Properties → List identifier:**) sends datagrams to the same global network variables list (GNVL) of any controller. If the **List Identifier** is not unique, this can cause an interruption in the exchange of variables.

<i>NOTICE</i>
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 a GNVL (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 *SoMachine, Network Variable Configuration, SE_NetVarUdp Library Guide*).

With the function block NETVARRESETERROR the detected NVL errors are reset and the communication is restarted.

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 a global variables list (GVL) is created. In the receiver controller the corresponding global network variables list (GNVL) 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:

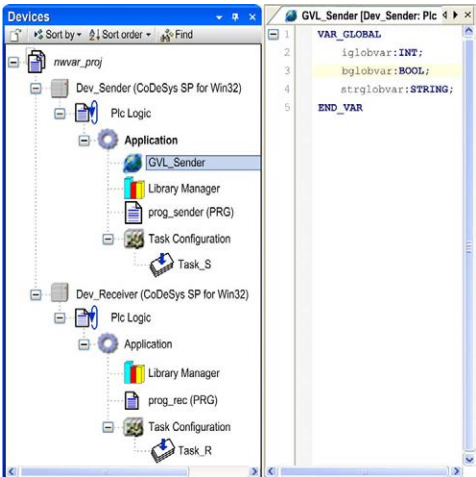
- Create a POU (program) **prog_sender** below the **Application** node of **Dev_Sender**.
- Under the **Task Configuration** node of this application, add the task **Task_S** that calls **prog_sender**.
- Create a POU (program) **prog_rec** below the **Application** node of **Dev_Receiver**.
- Under the **Task Configuration** node of this application, add the task **Task_R** that calls **prog_rec**.

NOTE: The 2 controllers must be configured in the same subnet of the Ethernet network.

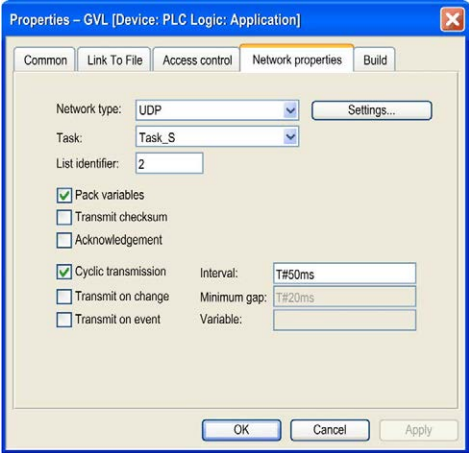
Defining the GVL for the Sender

Step 1: Define a global variable list in the sender controller:

Step	Action	Comment
1	In the Devices pane, right-click the Application node of the controller Dev_Sender and select the commands Add Object → Global Variable List...	The Add Global Variable List dialog box is displayed.
2	Enter the Name GVL_Sender and click Open to create a new global variable list.	The GVL_Sender node appears below the Application node in the Devices pane 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	—

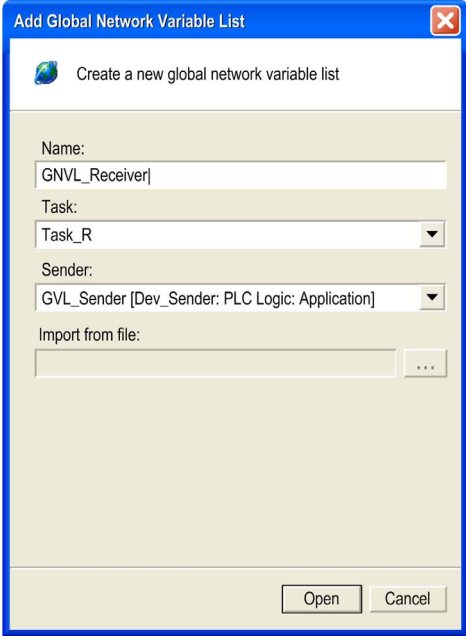


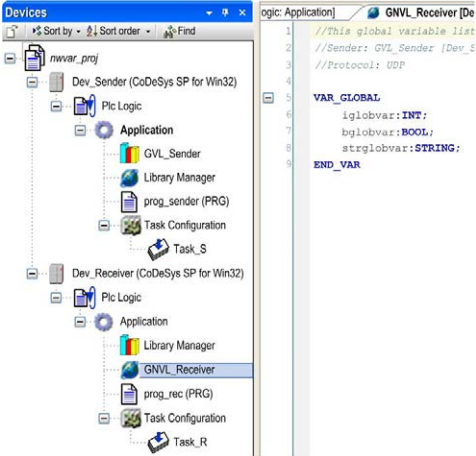
Step 2: Define the network properties of the sender GVL:

Step	Action	Comment
1	In the Devices pane, right-click the GVL_Sender node and select the command Properties....	The Properties - GVL_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 GVL network properties are set.

Defining the GNVL for the Receiver

Step 1: Define a global network variable list in the receiver controller:

Step	Action	Comment
1	In the Devices pane, right-click the Application node of the controller Dev_Receiver and select the commands 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 GVL defined for the sender controller.

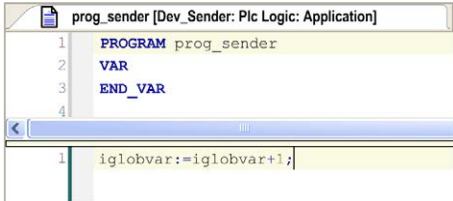
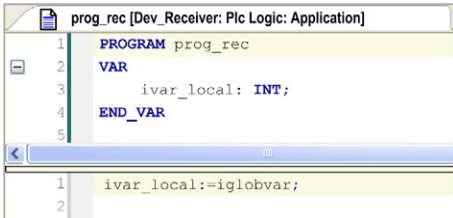
Step	Action	Comment
3	Click Open .	The dialog box is closed and the GNVL_Receiver appears below the Application node of the Dev_Receiver controller: 

This GNVL automatically contains the same variable declarations as the **GVL_Sender**.

Step 2: View and / or modify the network settings of the GNVL:

Step	Action	Comment
1	In the Devices pane, 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 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 a reading global network variables list (GNVL) 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:

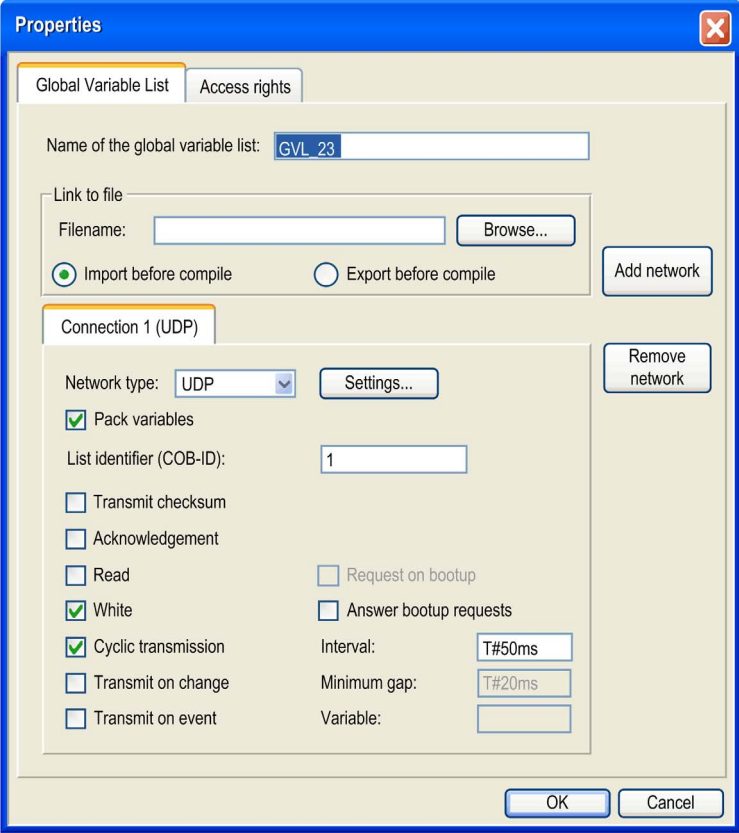
Step	Action	Comment
1	Re-create the network variables list (NVL) that is already available in the earlier version (V2.3) in the latest version (V3.x).	To achieve this, add a global variables list (GVL) with network properties, containing the same variables declarations as in the NVL of the earlier version (V2.3).
2	Export the new GVL to a *.exp file by using the Link to File tab.	NOTE: Activate the option Exclude from build in the Build tab to keep the GVL in the project without getting precompile events and ambiguous names. Deactivate the option to re-create the *.exp file again in case any modifications on the GVL are required.
3	Re-import the list.	To achieve this, create a new global network variables list (GNVL) by using the previously generated *.exp file in order to create an appropriately configured receiver list.

These steps are illustrated by the following example.

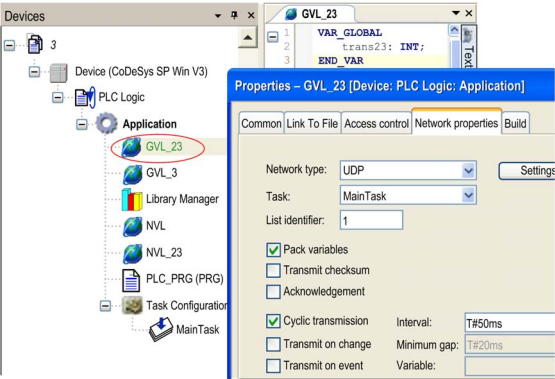
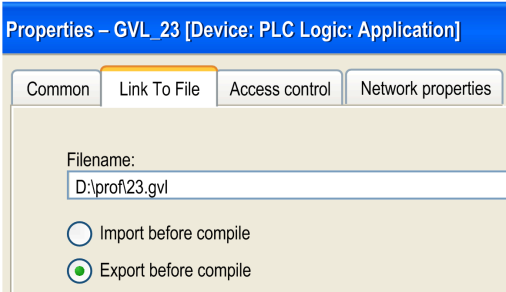
Example

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 <i>23.pro</i> contains a global variables list GVL_23 with the following declaration: <pre>VAR_GLOBAL trans23:INT; END_VAR</pre>
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: <pre>VAR_GLOBAL trans23:INT; END_VAR</pre> 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 a GVL object named GVL_23 to an application.	–
2	Set the network properties as defined within the <i>23.pro</i> project.	
3	In the Link to File tab, configure a target export file <i>23.gvl</i> .	
4	In the Build tab, set the Exclude from Build option.	This setting allows you to keep the file on disk for later modifications. 

Step	Action	Comment
5	Compile the project.	The <i>23.gv</i> file is generated and contains variable and configuration settings: <pre> GVVL> <Declarations><![CDATA[VAR_GLOBALB trans23: INT; 0 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> </GVVL> </pre>
6	Add a GNVL object in the V3.x project from the <i>23.gv</i> /export file (with the Import from file: command).	This serves to read the variable <code>trans23</code> 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 <code>trans23</code> from project <i>23.pro</i>.

Chapter 19

Task Editor

What Is in This Chapter?

This chapter contains the following topics:

Topic	Page
Information on the Task Configuration	420
Properties Tab	421
Monitor Tab	422
Configuration of a Specific Task	424
Task Processing in Online Mode	427

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 appropriate 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 ([see page 424](#)) 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 ([see page 424](#)) 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 ([see page 427](#)) the tasks will be executed.

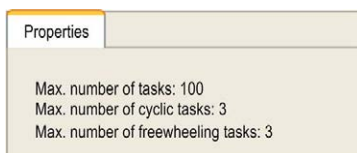
For each task, you can configure a time control (watchdog). The possible settings depend on the specific controller platform.

Properties Tab

Overview

When the **Task Configuration** (*see page 420*) node is selected, the **Properties** tab will be opened in the task editor view.

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.

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, the number of the calls and the code cover of the POUs, which are controlled by a task. In online mode, the task processing can be monitored.

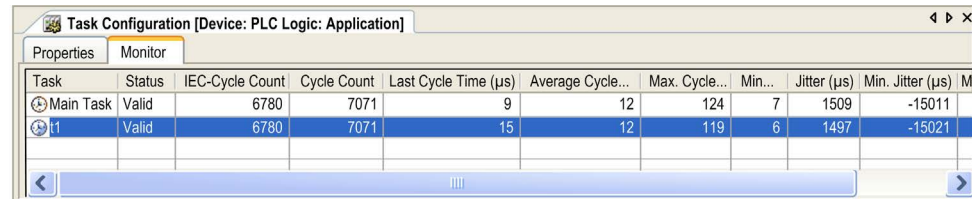
Online View of the Task Editor

When you select the top node in the **Task Configuration** tree, besides the **Properties** tab (see page 421), 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 (see page 421) 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, Monitoring



Task	Status	IEC-Cycle Count	Cycle Count	Last Cycle Time (µs)	Average Cycle...	Max. Cycle...	Min...	Jitter (µs)	Min. Jitter (µs)	M
Main Task	Valid	6780	7071	9	12	124	7	1509	-15011	
t1	Valid	6780	7071	15	12	119	6	1497	-15021	

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)	Last measured jitter* in μs
Min. Jitter (μs)	Minimum measured jitter* in μs
Max. Jitter (μs)	Maximum measured jitter* in μs
* jitter: Time that passes after the task has been started until the operating system indicates that it is running.	

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 context menu.

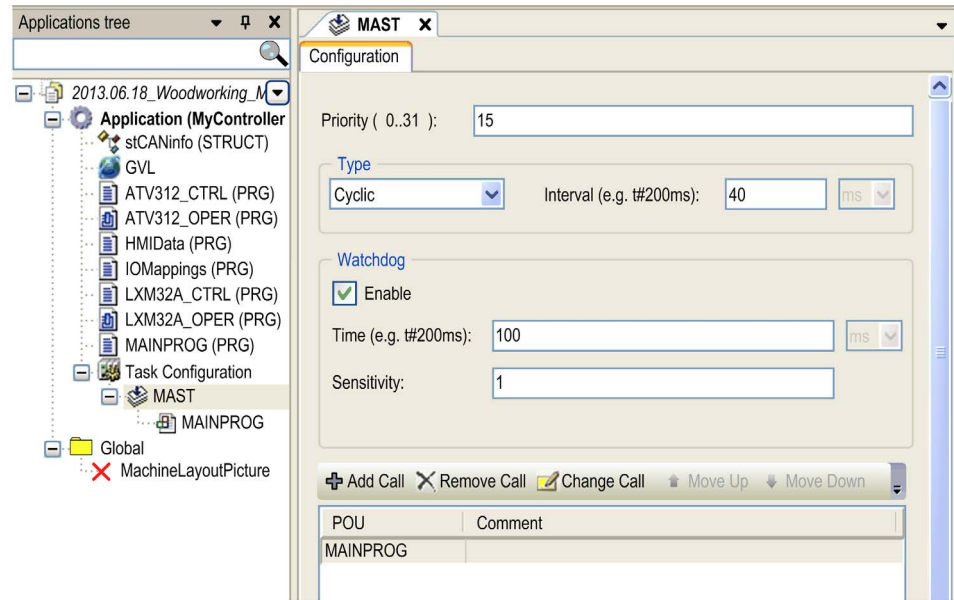
Configuration of a Specific Task

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.

Configuration tab for a 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

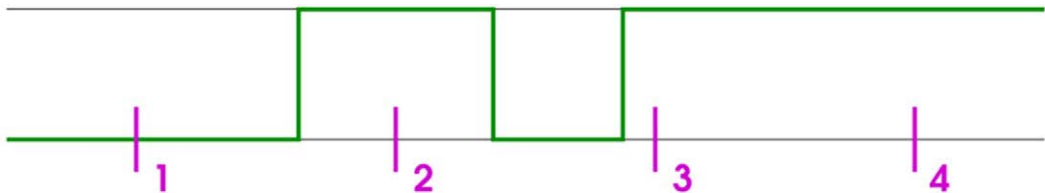
Type	
The selection list offers the following task types:	
Cyclic	The task will be processed cyclic according to the time definition (task cycle time) given in the field Interval (see below).

Type	
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.
Status	The task will be started if the variable defined in the Event field is TRUE. NOTE: This function is not available for all supported controllers. For more information, refer to the Programming Guide of your controller.
Event	The task will be started as soon as the variable defined in the Event field gets a rising edge.
External event	The task will be started as soon as the system event, which is defined in the 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.)

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

Obligatory Entries Depending on Task Choice

Entry	Description
Interval task cycle time	Obligatory for type Cyclic. The time (in milliseconds [ms]) after which the task should be restarted. NOTE: Consider the currently used bus system when setting the task cycle time. For example on a CAN bus, you can set the Bus cycle task in the CANopen I/O Mapping tab. It must correspond to the currently set transmission rate and the number of frames used on the bus. Further on the times set for heartbeat, nodeguarding, and sync always should be a multiple of the task cycle time. Otherwise CAN frames can get lost. For further information, refer to the <i>Device Editor</i> part of the SoMachine online help.
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.

Watchdog Settings

For each task, you can configure a time 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 detected if the execution time of the task exceeds the defined task time limit (**Time**) relative to the defined **Sensitivity**. If option **Update IO while in stop** is enabled in the controller settings dialog box, the outputs will be set to the pre-defined default values depending upon the particular controller platform.

Time (e.g. t#200ms)	Defines the allowable maximum execution time for a task. When a task takes longer than this, the controller will report a task watchdog exception.
Sensitivity	Defines the number of task watchdog exceptions that must occur before the controller detects an application error.

NOTE: The watchdog function is not available in simulation mode.

For further information, refer to the *System and Task Watchdog* chapter of the Programming Guide of your controller.

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 currently 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

Which Task Is Being Processed?

For the execution of the tasks defined in the **Task Configuration**, the following rules apply:

- That task is executed, whose condition has been met. This means, if the specified time has expired, or after its condition (event) variable exhibits a rising edge.
- If several tasks have a valid requirement, then the task with the highest **Priority** will be executed.
- If several tasks have valid conditions and equivalent priorities, then the task with the longest waiting time will be executed first.
- 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**.

Chapter 20

Watch List Editor

What Is in This Chapter?

This chapter contains the following topics:

Topic	Page
Watch View / Watch List Editor	430
Creating a Watch List	431
Watch List in Online Mode	433

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 (*see page 433*) 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 (*see page 431*).

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 (*see page 434*) automatically with all currently forced values of the active application.

Creating a Watch List

Overview

To set up a watch list **Watch<n>** in the **Watch** view, click in a field of the **Expression** column and press the SPACE key to edit the column **Expression**. Enter the complete path for the desired watch expression. The **Input Assistant** is available via button

Syntax for Watch Expression

<device name>.<application name>.<object name>.<variable name>

Example

Example:

Dev1.App1.PLC_PRG.ivar

The type of the variable is indicated by an icon before the variable name:

Icon	Variable
	Input
	Output
	Normal

After you have entered the variable in the **Expression** column, the corresponding data type is added automatically in the column **Type**. If a comment has been added to the declaration of a variable, it is added in the column **Comment**.

The column **Value** displays the current value of the expression in online mode (*see page 433*).

To prepare a value for a variable, click in the assigned field of the column **Prepared value** and directly enter the desired value.

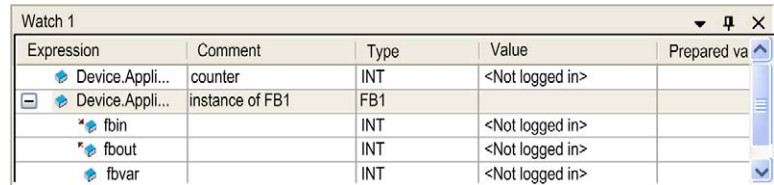
In case of a boolean variable the handling is even easier: You can toggle boolean preparation values by use RETURN or SPACE according to the following order:

If the value is TRUE, the preparation steps are FALSE -> TRUE -> no entry; else, if the value is FALSE, the preparation steps are TRUE -> FALSE -> no entry.

Do the same for the desired further expressions/variables in further lines. See an example in the next image which shows the watch view in offline mode: It contains expressions of objects PLC_PRG and Prog_St.

Keep in mind that in case of a structured variable, as with the function block instance, the particular instance components automatically get added when you enter the instance name (see in the example: Dev1.App1.PLC_PRG.fbinst). Click the plus-/minus sign to display or hide them in a fold.

Example, watch view in offline mode



Expression	Comment	Type	Value	Prepared va
Device.Appli...	counter	INT	<Not logged in>	
Device.Appli...	instance of FB1	FB1		
fbinst		INT	<Not logged in>	
fbout		INT	<Not logged in>	
fbvar		INT	<Not logged in>	

In online mode ([see page 433](#)), you can use the list for monitoring of the variable values.

NOTE: In online mode you can add expressions to the watch list by use of the command **Add Watch**.

Watch List in Online Mode

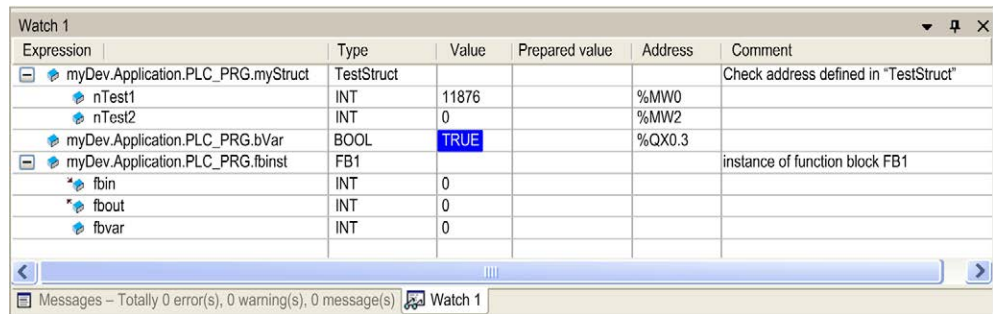
Monitoring

A watch list (*see page 430*) (**Watch<n>**) in online mode shows the current value of a variable in the **Value** column. This is the value the variable has between 2 task cycles.

Also a possibly 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 (*see page 387*).

See chapter *Creating a Watch List* (*see page 431*) 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



Expression	Type	Value	Prepared value	Address	Comment
myDev.Application.PLC_PRG.myStruct	TestStruct				Check address defined in "TestStruct"
nTest1	INT	11876		%MW0	
nTest2	INT	0		%MW2	
myDev.Application.PLC_PRG.bVar	BOOL	TRUE		%QX0.3	
myDev.Application.PLC_PRG.fbinst	FB1				instance of function block FB1
fbIn	INT	0			
fbOut	INT	0			
fbVar	INT	0			

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).

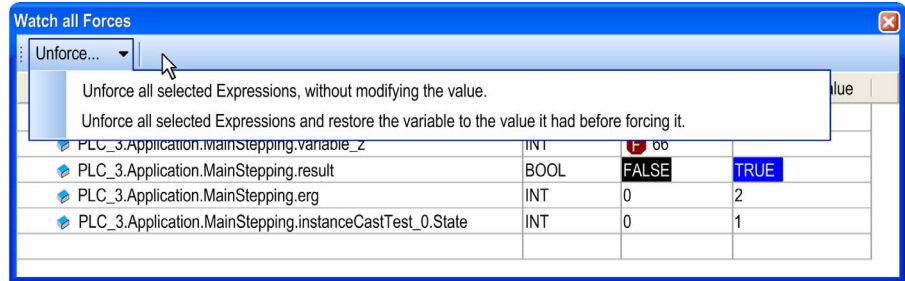
Watch All Forces

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 1 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



Chapter 21

Tools Within Logic Editors

What Is in This Chapter?

This chapter contains the following topics:

Topic	Page
Function and Function Block Finder	436
Input Assistant	440

Function and Function Block Finder

Overview

SoMachine 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 SoMachine 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 context menu

The **FFB Finder** dialog box opens:

FFB Finder

Find what: Find

(You can use * and ? as wildcards)

Company: ▼

☐ Match case ☐ Include comments in search ☐ Search in project libraries only

Results

Loaded	Name	Library	Version	Company	Comment
☐	SCHNEIDER_M258_ENCODER_REF	LMC058 Motion	1.0.1.2	Schneider Electric	
☐	HSCMain_M258	M258 Expert IO	1.0.1.14	Schneider Electric	
☐	ENCODER_M258	M258 Expert IO	1.0.1.14	Schneider Electric	
☐	PWM_M258	M258 Expert IO	1.0.1.14	Schneider Electric	
☐	FrequencyGenerator_M258	M258 Expert IO	1.0.1.14	Schneider Electric	
☐	HSCSimple_M258	M258 Expert IO	1.0.1.14	Schneider Electric	

6 occurrences in 2 libraries found Help

FUNCTION_BLOCK **FrequencyGenerator_M258**

FrequencyGenerator_M258

Description

This function block commands a square wave signal output at the specified frequency.

Notes:

The function block instance name must match the name defined by configuration.

All hardware-related information managed by this FB is synchronized with the MAST task cycle. Therefore:

- use the FB instance in the MAST task.

OK Cancel

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 following companies from the Company list: ● 3S - Smart Software Solutions GmbH ● CAA Technical Workgroup ● Schneider Electric ● System 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 SoMachine 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.

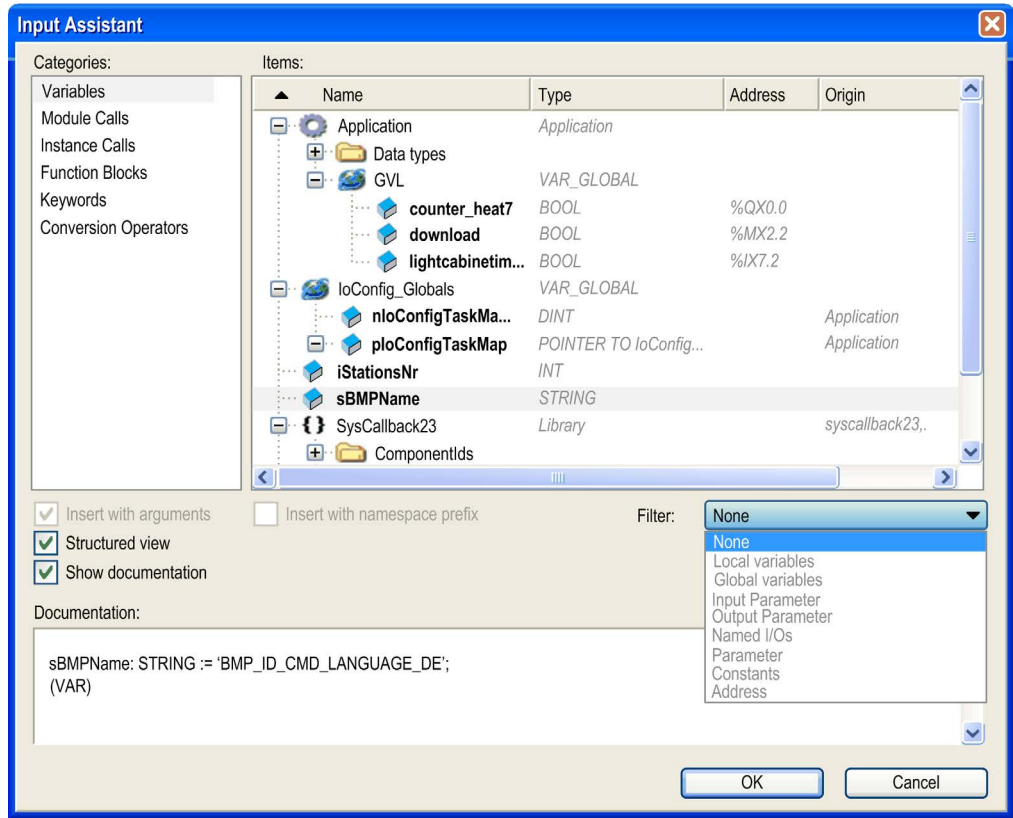
Repeat this operation whenever you need assistance in finding a specific function / function block.

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.

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: <pre>fb1(fb1_in:= , fb1_out=>)</pre>
Insert with namespace prefix	If this option is selected, the item is inserted with the prefixed namespace. Currently this option is only available for global variables.

Part VI

Tools

What Is in This Part?

This part contains the following chapters:

Chapter	Chapter Name	Page
22	Data Logging	445
23	Recipe Manager	447
24	Trace Editor	465
25	Symbol Configuration Editor	491
26	SoMachine Controller - HMI Data Exchange	499

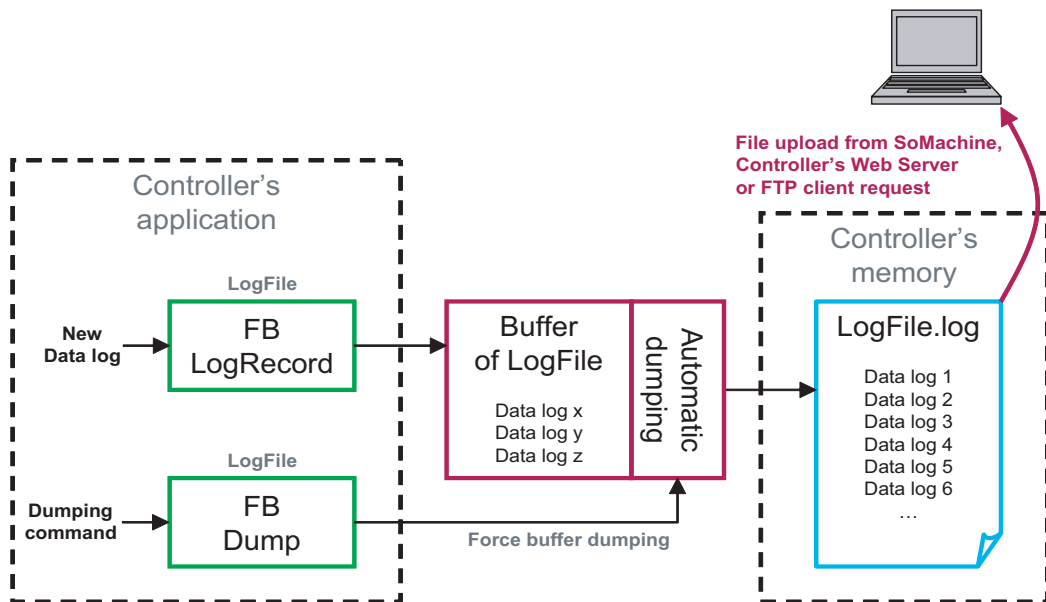
Chapter 22

Data Logging

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 SoMachine or by the controller web server.

NOTE: Only controllers with file management functionality can support data logging. Refer to your controller programming manual to see if it supports file management. The software itself does not evaluate your controller for compatibility with data logging activities.

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.

Chapter 23

Recipe Manager

What Is in This Chapter?

This chapter contains the following topics:

Topic	Page
Recipe Manager	448
Recipe Definition	452
RecipeMan Commands	456

Recipe Manager

Overview

The **Recipe Manager** functionality is only available if selected in the currently used feature set (**Options → Features → Predefined feature sets**).

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 for setting and watching control parameters 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 (*see page 456*).

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 SoMachine 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* ([see page 452](#)).

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).

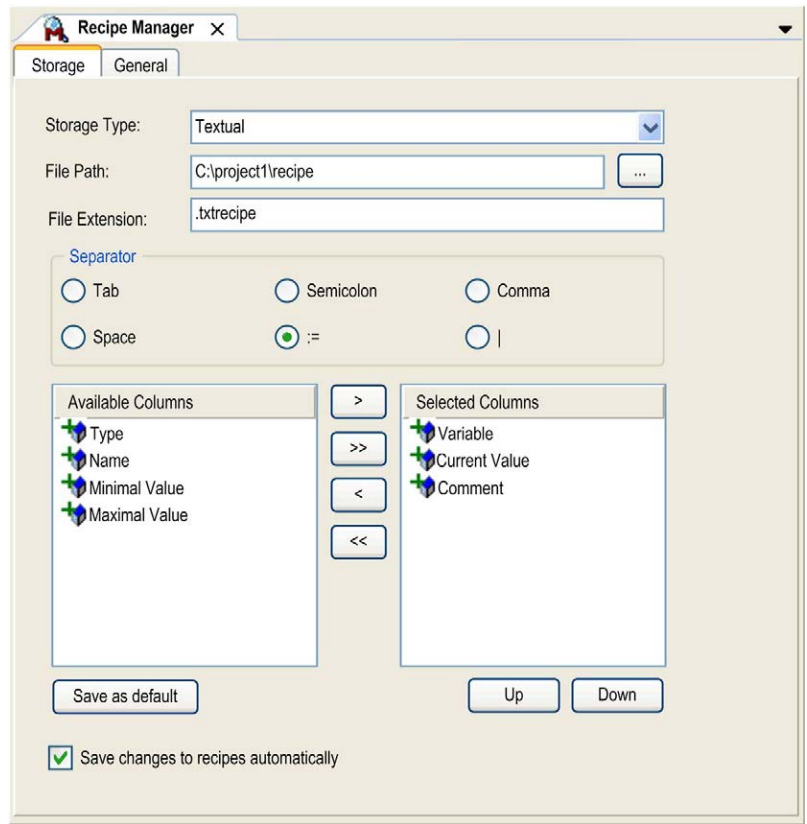
Recipe Management Objects in the Tools Tree

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* ([see page 452](#)).

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 will be stored.
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* 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 (see page 451).
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.
Save changes to recipes automatically	Select this option to update immediately the storage files after any modification of a recipe during run time.

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. This avoids downloading the manager. 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.

Recipe Definition

Overview

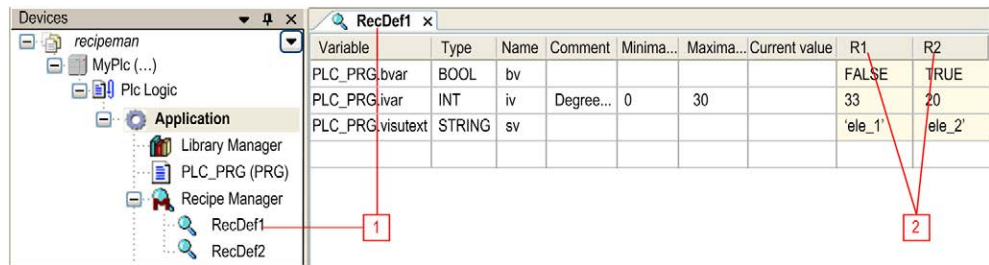
The recipe manager (*see page 448*) handles 1 or several recipe definitions. A recipe definition contains a list of variables and 1 or several recipes (value sets) for these variables. By using different recipes, you can assign another set of values to a set of variables on the controller in one stroke. There is no limitation of the number of recipe definitions, recipes, and variables per recipe.

Recipe Definition

You can add 1 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 will be titled with the name of the recipe definition.

Parameter	Description
Variable	In a 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 <code>plc_prg.ivar</code> . Click the ... button to open the input assistant.
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.

Parameter	Description
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 changes to recipes automatically	It is a best practice to activate this option because it affects the usual behavior of a recipe management: the storage files will be 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 DEL 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 currently selected line. In doing so, recipe values will be inserted in the matching recipe column, if available.

To add a recipe to the recipe definition, execute the **Add a new recipe** command (*see SoMachine, Menu Commands, Online Help*) when the focus is in the editor view. For each recipe, an own column will be 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.

The following methods are supported:

- ReadRecipe: The current variable values are taken into the recipe.
- WriteRecipe: The recipe is written into the variables.
- SaveRecipe: The recipe is stored into a standard recipe file.
- LoadRecipe: The recipe is loaded from a standard recipe file.
- CreateRecipe: A new recipe in the recipe definition is created.
- DeleteRecipe: An existing recipe from a recipe definition is deleted.

See in the following paragraphs, how the recipes behave in the particular online states. It is a best practice to set the option **Save changes to recipes 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 SoMachine, Menu Commands, Online Help*) and **Remove recipes** (*see SoMachine, 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 will be 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.library`, in the application code, or via inputs on visualization elements.

Recipe handling in online mode if **Save changes to recipes 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 current project.	Dynamically created recipes remain unchanged.
Online Reset Origin	The application will be 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 <code>RecipeManCommands</code> function block.	
Stop	At a stop/start of the controller, the recipes remain unchanged.	

Recipe handling in online mode if **Save changes to recipes 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 current project. However, these are only set in the memory. In order to store the recipe in a file, the save command must be used explicitly.	Dynamically created recipes get lost.
Online Reset Origin	The application will be 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 <code>RecipeManCommands</code> 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* ([see page 450](#)).
- For a description of the particular RecipeManCommands methods ([see page 456](#)), refer to the documentation within the library.
- For the appropriate input configuration of a visualization element, refer to its help page (category **Input** → **execute command**).

The following actions on recipes are possible:

Actions	Description
Create recipe (= Add a new recipe)	A new recipe will be created in the specified recipe definition.
Read recipe	The current values of the variables of the specified recipe definition will be read from the controller and be written to the specified recipe. This means that the values will be stored implicitly (in a file on the controller). They will also will 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, will be written to the variables on the controller.
Save Recipe	The values of the specified recipe will be written to a file with extension <i>*.txtrecipe</i> , the name of which you have to define. For this purpose, the standard dialog box for saving a file in the local file system will open. NOTE: The implicitly used recipe files, necessary as a buffer for reading and writing of the recipe values, may not get overwritten. This means that the name for the new recipe file must be different from <recipe name>.<recipe definition name>.txtrecipe.
Load Recipe	The recipe which has been stored in a file (see the Save Recipe description) can be reloaded from this file. The standard dialog box for browsing for a file will open for this purpose. The filter is automatically set to extension <i>*.txtrecipe</i> . After reloading the file, the recipe values will be updated accordingly in the recipe manager.
Delete recipe (= Remove recipe)	The specified recipe will be removed from the recipe definition.
Change recipe	The value of the project variables can be changed. With a following write recipe action, the appropriate project variables are written with the new values.

Creating Specific Tasks for Recipe Management

Using recipe files (create, read, write, delete) may have impact on the performance of the logic controllers. If you wish to use recipe files, consider creating specific tasks ([see page 424](#)) with a low priority and with the **Watchdog** function disabled.

RecipeMan Commands

Overview

When calling a recipe command, internal data access will be performed. Depending on the device type, this will take 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 changes to recipes automatically** will also perform a file access with each change of the recipe. Deactivate this option if the storage of the recipe is triggered by the application.

Return Values

The following return values are possible for recipe commands:

Return Value	Description
ERR_NO_RECIPE_MANAGER_SET	No recipe manager is available on the controller.
ERR_RECIPE_DEFINITION_NOT_FOUND	The recipe definition does not exist.
ERR_RECIPE_ALREADY_EXIST	The recipe already exists in the recipe definition.
ERR_RECIPE_NOT_FOUND	The recipe does not exist in the recipe definition.
ERR_RECIPE_FILE_NOT_FOUND	The recipe file does not exist.
ERR_RECIPE_MISMATCH	The content of the recipe file does not match the current recipe. NOTE: This return value is only generated when the storage type is textual and when a variable name in the file does not match the variable name in the recipe definition. The recipe file is not loaded.
ERR_RECIPE_SAVE_ERR	The recipe file could not be opened with write access.
ERR_FAILED	The operation was unsuccessful.
ERR_OK	The operation was successful.

CreateRecipe

This method creates a new recipe in the specified recipe definition and afterwards reads the current controller values into the new recipe. At the end, the new recipe is stored in the default file.

Parameter	Description
RecipeDefinitionName:	name of the recipe definition
RecipeName:	name of the recipe

Return values (*see page 456*):

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 current controller values into the new recipe.

Parameter	Description
RecipeDefinitionName:	name of the recipe definition
RecipeName:	name of the recipe

Return values (*see page 456*):

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	Description
RecipeDefinitionName:	name of the recipe definition
RecipeName:	name of the recipe

Return values (*see page 456*):

ERR_NO_RECIPE_MANAGER_SET, ERR_RECIPE_DEFINITION_NOT_FOUND, ERR_RECIPE_NOT_FOUND, ERR_FAILED, ERR_OK

DeleteRecipeFile

This method deletes the standard recipe file from a recipe.

Parameter	Description
RecipeDefinitionName:	name of the recipe definition
RecipeName:	name of the recipe

Return values (*see page 456*):

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 standard recipe file and afterwards writes the recipe into the controller variables.

Parameter	Description
RecipeDefinitionName:	name of the recipe definition
RecipeName:	name of the recipe

Return values (*see page 456*):

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	Description
RecipeDefinitionName:	name of the recipe definition
RecipeName:	name of the recipe
FileName:	name of the file

Return values (*see page 456*):

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 standard recipe file. The standard recipe file name is <recipe>.<recipe definition>.<recipeextension>.

Parameter	Description
RecipeDefinitionName:	name of the recipe definition
RecipeName:	name of the recipe

Return values (*see page 456*):

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

ReadAndSaveRecipe

This method reads the current controller values into the recipe and afterwards stores the recipe into the standard recipe file.

Parameter	Description
RecipeDefinitionName:	name of the recipe definition
RecipeName:	name of the recipe

Return values (*see page 456*):

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 current 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	Description
RecipeDefinitionName:	name of the recipe definition
RecipeName:	name of the recipe
FileName:	name of the file

Return values (*see page 456*):

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 standard recipe file. The content of an existing file would be overridden. The standard recipe file name is <recipe>.<recipedefinition>.<recipeextension>.

Parameter	Description
RecipeDefinitionName:	name of the recipe definition
RecipeName:	name of the recipe

Return values (*see page 456*):

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 current controller values into the recipe.

Parameter	Description
RecipeDefinitionName:	name of the recipe definition
RecipeName:	name of the recipe

Return values (*see page 456*):

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	Description
RecipeDefinitionName:	name of the recipe definition
RecipeName:	name of the recipe

Return values (*see page 456*):

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.

Parameter	Description
RecipeDefinitionName:	name of the recipe definition

Return values (*see page 456*):

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	Description
RecipeDefinitionName:	name of the recipe definition

Return values: -1 : if the recipe definition is not found.

GetRecipeNames

This method returns the recipe names from the corresponding recipe definition.

Parameter	Description
RecipeDefinitionName:	name of the recipe definition
pStrings :	the strings where the recipe values should be stored
iSize :	the size of an array of string
iStartIndex :	the start index can be used for a scrolling function

Return values ([see page 456](#)):

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 string:

```
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	Description
RecipeDefinitionName:	name of the recipe definition
RecipeName	name of the recipe
pStrings :	the strings where the recipe values should be stored
iSize :	the size of an array of string
iStartIndex :	the start index can be used for a scrolling function
iStringLength :	the length of the string in the array

Return values (see page 456):

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 string:

```
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	Description
RecipeDefinitionName:	name of the recipe definition
RecipeName	name of the recipe
pStrings :	the strings where the recipe values should be stored
iSize :	the size of an array of string
iStartIndex :	the start index can be used for a scrolling function

Return values (see page 456):

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 string:

```
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	Description
RecipeDefinitionName:	name of the recipe definition
RecipeName	name of the recipe
pStrings :	the strings where the recipe values should be stored
iSize :	the size of an array of string
iStartIndex :	the start index can be used for a scrolling function

Return values ([see page 456](#)):

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 string:

```
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 returns the last detected error of the previous operations.

Return values ([see page 456](#)): ERR_NO_RECIPE_MANAGER_SET, ERR_OK

ResetLastError

This method resets the last detected error.

Return values ([see page 456](#)): ERR_NO_RECIPE_MANAGER_SET, ERR_OK

Chapter 24

Trace Editor

What Is in This Chapter?

This chapter contains the following sections:

Section	Topic	Page
24.1	Trace Object	466
24.2	Trace Configuration	472
24.3	Trace Editor in Online Mode	488
24.4	Keyboard Operations for Trace Diagrams	489

Section 24.1

Trace Object

What Is in This Section?

This section contains the following topics:

Topic	Page
Trace Basics	467
Creating a Trace Object	469

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 SoMachine buffer of a specified size. They can be observed in the form of a two-dimensional graph plotted as a function of time.

Way of Tracing Data

The tracing of data on the control is performed in 2 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**).

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 ([see page 473](#)) and to the record (trigger) settings ([see page 477](#)).
- 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 SoMachine, Menu Commands, Online Help](#)).
- trace system variables of other system components.

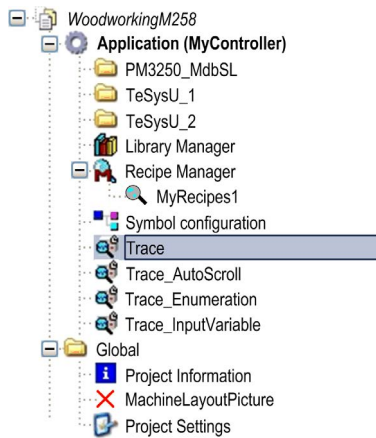
Furthermore, the additional command ([see SoMachine, Menu Commands, Online Help](#)) **Online List** is available.

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 ([see page 578](#)) 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 ([see page 470](#)). 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**.

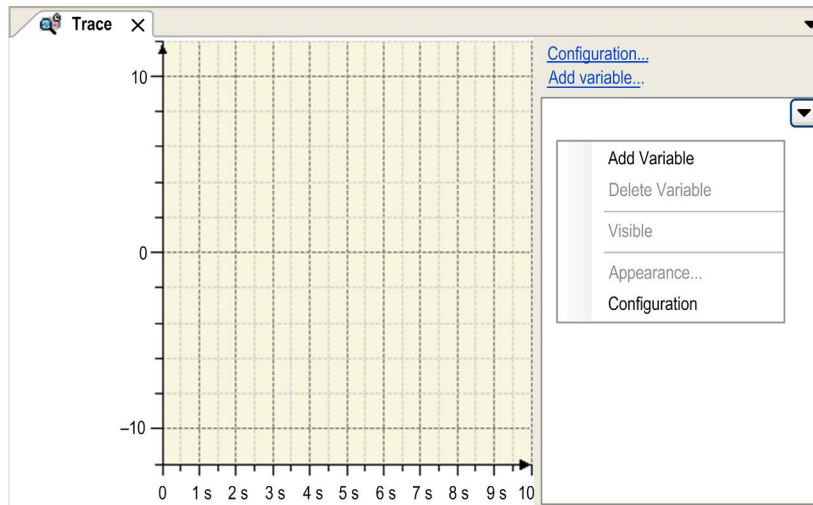
Creating a Trace Object

Overview

To insert a trace object in the **Tools tree**, select the **Application** node, click the green plus button and execute the command **Add other objects → Trace....** Double-click the **Trace** node in the **Tools tree** to open the trace editor.

Configuration

New created trace with context 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.

To configure or change the trace settings, use the commands of the context menu in the trace tree area:

- **Add Variable....**: Opens the **Trace Configuration** dialog box with **Variable Settings** (see page 473).
- **Delete Variable**: 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.

- **Appearance ...:** The **Edit Appearance** dialog box opens ([see page 483](#)). It allows you to configure the appearance of the graph and the coordinate system. This command is grayed out until a configuration is loaded.
- **Configuration...:** Opens the **Trace Configuration** dialog box with **Record Settings** ([see page 477](#)).

Features

For running the trace, use the following commands:

- **Add variable** ([see SoMachine, Menu Commands, Online Help](#))
- **Download Trace** ([see SoMachine, Menu Commands, Online Help](#))
- **Start/Stop Trace** ([see SoMachine, Menu Commands, Online Help](#))
- **Reset Trigger** ([see SoMachine, Menu Commands, Online Help](#))

For customizing the view of the graphs, use the following commands:

- **Cursor** ([see SoMachine, Menu Commands, Online Help](#))
- **Mouse Zooming** ([see SoMachine, Menu Commands, Online Help](#))
- **Reset View** ([see SoMachine, Menu Commands, Online Help](#))
- **Auto Fit** ([see SoMachine, Menu Commands, Online Help](#))
- **Compress** ([see SoMachine, Menu Commands, Online Help](#))
- **Stretch** ([see SoMachine, Menu Commands, Online Help](#))
- **Multi Channel** ([see SoMachine, Menu Commands, Online Help](#))
- For further information, refer to the chapter ([see page 489](#)) *Keyboard Operations for Trace Diagrams*.

For access to traces stored on the runtime system, use the following commands:

- **Online List** ([see SoMachine, Menu Commands, Online Help](#))
- **Upload Trace** ([see SoMachine, Menu Commands, Online Help](#))

For access to traces stored on the disc, use the following commands:

- **Save Trace...** ([see SoMachine, Menu Commands, Online Help](#))
- **Load Trace...** ([see SoMachine, Menu Commands, Online Help](#))
- **Export symbolic trace config** ([see SoMachine, 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 will be 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* ([see page 488](#)).

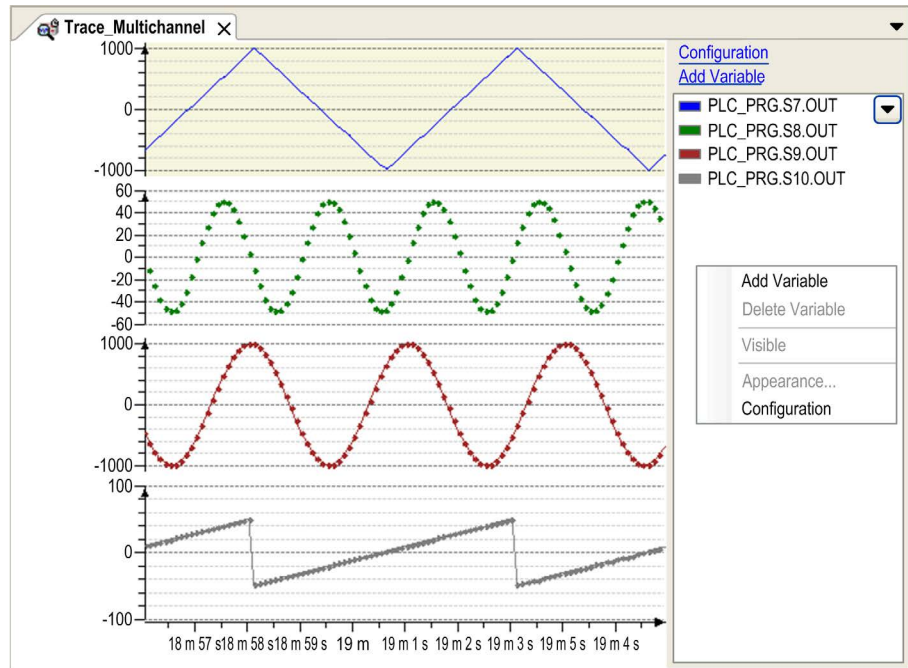
Step	Action
1	Login and run the associated application. Result: The application runs on the controller.
2	Download trace Result: The trace graphs are immediately displayed according to the trace configuration.

Step	Action
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



Section 24.2

Trace Configuration

What Is in This Section?

This section contains the following topics:

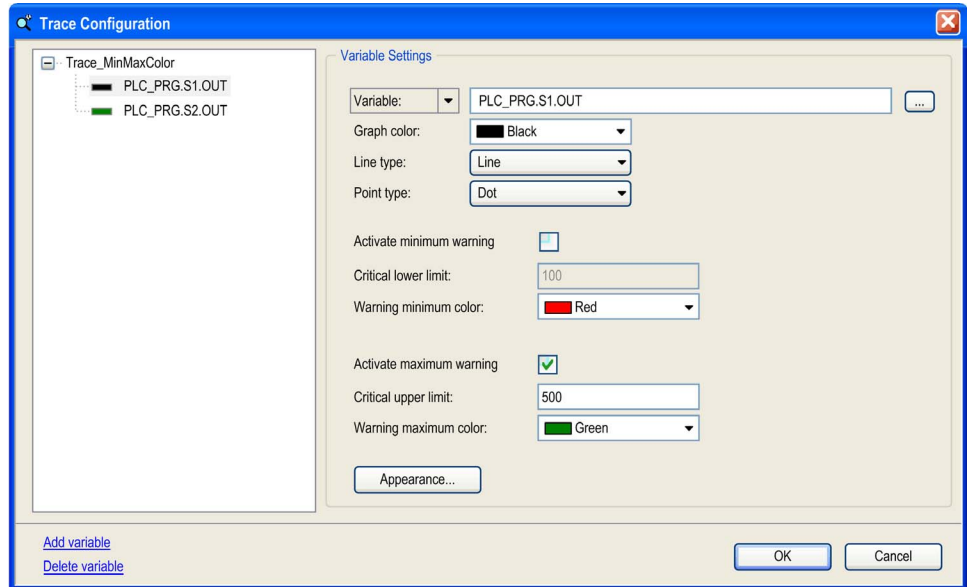
Topic	Page
Variable Settings	473
Record Settings	477
Advanced Trace Settings	481
Edit Appearance	483
Appearance of the Y-axis	487

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 Variable	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 current 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.

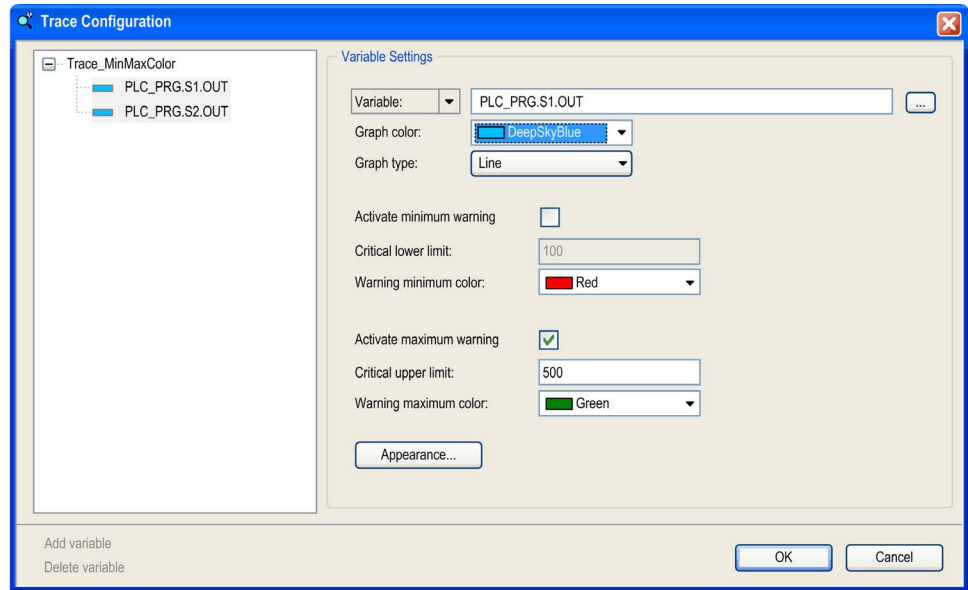
Parameter		Description
Variable		Enter the name (path) of the signal to specify the signal that will be traced. A valid signal is an IEC variable, a property, a reference, the contents of a pointer, or an array element of the application. Allowed types are all IEC basic types, except STRING, WSTRING 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 current 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 (<i>see page 168</i>) that is used as a trace (or trigger) variable must get the compiler attribute Attribute Monitoring (<i>see page 578</i>).
Graph color		Select a color from the color selection list in which the trace curve for the variable will be displayed.
Line type		Specify the way samples will be 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 will be 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 will be displayed in the color defined in Warning minimum color as soon as the variable exceeds the value defined in Critical lower limit .

Parameter	Description
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.
Activate Maximum Warning	If this option is activated, the trace graph will be 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.
Appearance...	Opens the Appearance of the Y-axis dialog box. It allows you to set up the display of the trace window for the currently configured Y-axis (colors and scroll behavior) for every variable in its own style. These settings are used when the trace diagram is displayed in multi-channel view.

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 changes in the dialog box **Variable Settings** apply for all selected variables. The same can be done with SHIFT + ARROW UP/DOWN or CTRL + ARROW UP/DOWN.

Multi-selection in dialog box **Trace Configuration**



Record Settings

Overview

The **Trace Configuration** dialog box with **Record Settings** opens if you execute the command **Configuration...** or if you double-click the trace name on the top of the trace tree. The configuration command is also available in the context 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

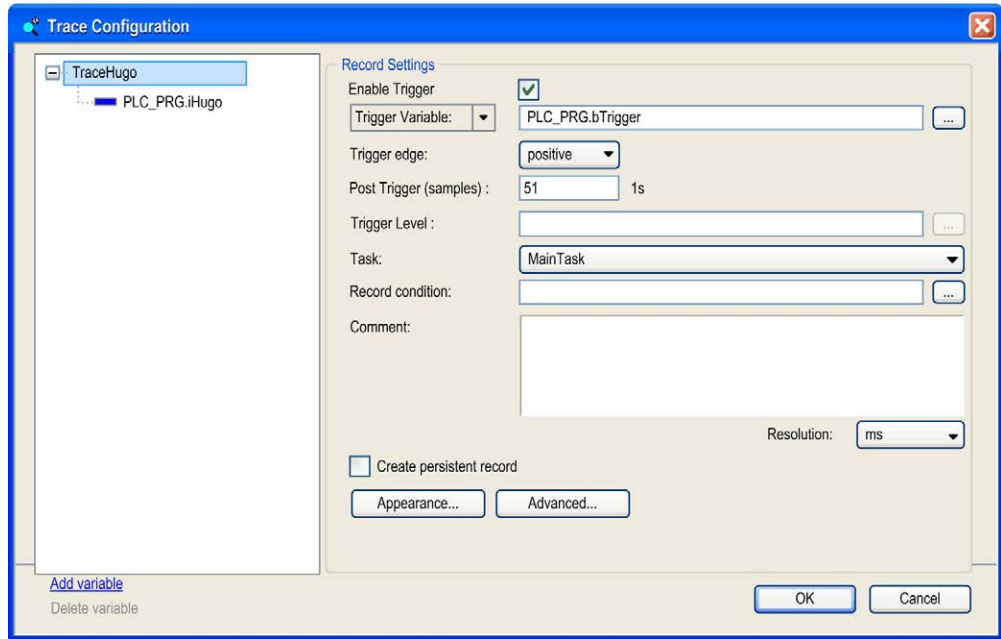
In most situations, it is not desired that tracing and displaying the input signals starts at random moments, such as immediately after the previous measurement, or when the user presses the start button. Most of the time it is preferred that the tracing is done when a trigger is fired for the configured 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

Trace Configuration dialog box with Record 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.

Parameter		Description
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. 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 (<i>see page 168</i>) that is used as a trace (or trigger) variable must get the compiler attribute Attribute Monitoring (<i>see page 578</i>).
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...(2³²-1)
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)

Parameter	Description
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 current 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, a time stamp with resolution in μs is recommended. This option is only possible when a trace manager is available in the controller.
Create persistent record	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.
Appearance...	Opens the Appearance of the Y-axis dialog box (see page 487). It allows you to set up the display of the trace window for the currently configured record, such as axes, colors, and scroll behavior.
Advanced...	Click this button to open the Advanced Trace Settings dialog box (see page 481). 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.

Advanced Trace Settings

Overview

The **Advanced Trace Settings** dialog box opens if you click the **Advanced...** button in the **Trace Configuration** dialog box with **Record Settings**.

Advanced Trace Settings dialog box

For all values given in records or cycles, the associated time span is shown right next to it (for example, **1m39s990ms**). The time span for buffers encloses the entire buffer. If the task is not set, not cyclical or a system task, then the task cycle time is unknown. In this case, the period cannot be calculated and will not be displayed on the right-hand side.

Description of the Parameters

Parameter	Description
Refresh rate (ms)	At this time distance, the captured data pairs (value with time stamp) of the tracing are stored in the buffer of the trace editor. Range: 150 ms...10000 ms Default value: 500 ms If the trace manager is used, then the data pairs are transferred at this time frame from the runtime system to the programming system. If the trace manager is not used, then the data are transmitted every 200 ms to the programming system.
Trace editor buffer size (samples)	Enter the buffer size of the trace in samples (records). This buffer has to be greater than or equal to twice the size of what the runtime buffer may be. Range: 1...10 ⁷

Parameter	Description
Measure in every n-th cycle	Select a time distance (in task cycles) between the capturing of the input signal from the list. Default and minimum value: 1 (this means a measure in every cycle)
Recommended runtime buffer size (samples)	The recommended number of samples of the runtime buffer for each trace signal is displayed. This value is computed-based on the task cycle time, the refresh time, and the value Measure in every n-th cycle . This means that one buffer is allocated for each trace variable. NOTE: The buffer size is given in samples and one buffer is created for each trace variable.
Override runtime buffer size	When this option is selected, the value entered here is used, instead of the default value of the runtime buffer size. Example: Range: 10...the trace editor buffer size

Edit Appearance

Overview

The **Edit Appearance** dialog box opens if you click the **Appearance...** button in the **Trace Configuration** dialog box with **Record Settings**.

The following settings define the appearance of the coordinate system and its X/Y-axis. The settings for the Y-axis are used when the trace diagram is displayed in single channel view. In multi-channel view, the settings done in the **Appearance of the Y-axis** dialog box are used.

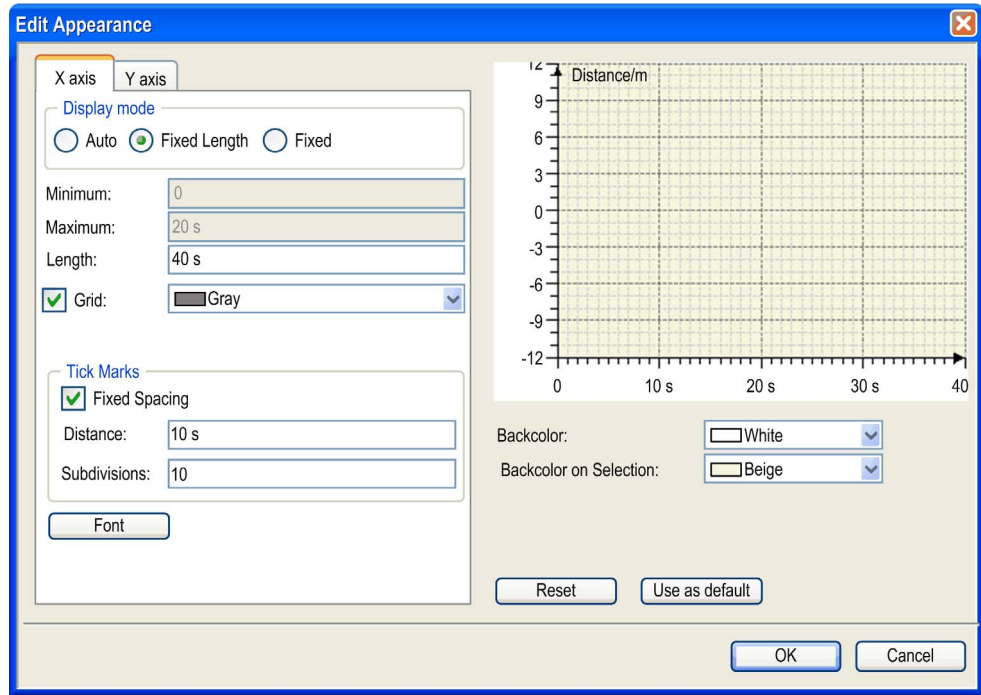
The settings for the X/Y-axis (done in the left part) and the coordinate system are immediately undertaken in the coordinate system displayed right.

You can manage the settings with the following buttons:

Button	Description
Reset	With this command, the appearance is reset to its default value.
Use as default	With this command, the current appearance is set as default. It will be used when a new trace or variable is configured.

X-axis Tab

X-axis tab of the **Edit Appearance** dialog box



Parameter		Description
Display mode		Select the display mode.
	Auto	If this option is activated, the time axis is automatically scaled according to the contents of the trace editor buffer (see page 481). The current contents of the trace buffer is displayed in the diagram. No further entries have to be set.
	Fixed length	If this option is activated, the displayed interval of the time axis has a fixed length. Define this length with the parameter Length . The scale is also adjusted to the length. The graph is automatically scrolled into a visible range. Therefore, a time interval with the configured length and the associated latest data is shown in the diagram. But only as many values as were recorded. As soon as new data are recorded, the display scrolls.
	Fixed	If this option is activated, the displayed interval of the X-axis is defined by the minimum and maximum value.
Minimum		This value defines the minimum displayed value of the time axis. ¹

Parameter		Description
Maximum		This value defines the maximum displayed value of the time axis. ¹
Length		This value defines the length of the displayed interval of the time axis. ¹
Grid		If this option is activated, a grid will be displayed. Select the color of the grid lines from the list.
Tick marks		–
	Fixed spacing	Select this check box to scale the axis in certain distances.
	Distance	Enter a positive distance. The X-axis is a time axis with default 1s.
	Subdivision	Enter a reasonable number of subdivisions for each distance.
Font		Opens the standard dialog box for defining the font for the trace display.

¹ The time entries do not need the prefix # as required in IEC code. Possible time entries are, for example, **2s**, **1ms** or **1m20s14ms**. Use us for microseconds, for example **122ms500us**. Useful values are also dependent on the resolution of the time axis.

Y-axis Tab

Y-axis tab of the **Edit Appearance** dialog box

Edit Appearance

X axis Y axis

Display mode

☐ Auto ☒ Fixed

Minimum: -12

Maximum: 12

☒ Grid: Gray

☒ Description: Distance/m

Tick marks

☒ Fixed spacing

Distance: 3

Subdivisions: 3

Font

Backcolor: White

Backcolor on Selection: Beige

Reset Use as default

OK Cancel

Preview: Distance/m vs time (0 to 40 s). Y-axis scale: -12 to 12. Grid lines are visible.

Parameter		Description
Display mode		Select the display mode.
	Auto	If this option is activated, the Y-axis is automatically scaled according to the captured values. No further entries have to be set.
	Fixed	If this option is activated, the displayed section of the Y-axis is defined by the minimum and maximum value.
Minimum		This value defines the minimum displayed value of the Y-axis.
Maximum		This value defines the maximum displayed value of the Y-axis.
Grid		If this option is activated, a grid will be displayed. Select the color of the grid lines from the list.
Description		If this option is activated, the Y-axis is labeled with the text entered in the field next to it.
Tick marks		–
	Fixed spacing	Select this check box to scale the axis in certain distances.
	Distance	Enter a positive distance. Default: 1
	Subdivision	Enter a number, from 1 to 10, of subdivisions for each distance.
Font		Click this button to open the standard dialog box for defining the font for the trace display.

Parameters of the Coordinate System

Parameter	Description
Backcolor	Choose the background color for the coordinate system from the list. It is used as long as the diagram is not selected in the trace window.
Backcolor on Selection	Choose the background color for the coordinate system from the list. It is used as long as the diagram is selected in the trace window.

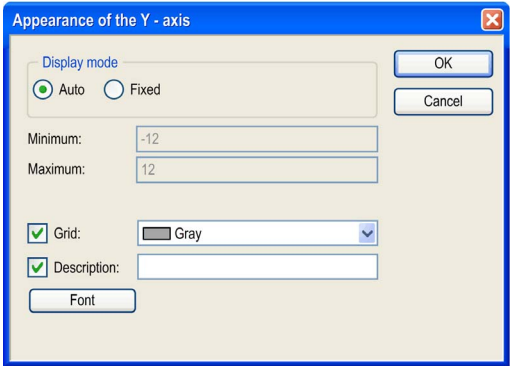
Appearance of the Y-axis

Overview

The **Appearance of the Y-axis** dialog box opens if you click the **Appearance...** button in the **Trace Configuration** dialog box with **Variable Settings**.

The following settings define the appearance of the Y-axis. They are used when the trace diagram is displayed in multi-channel view.

Appearance of the Y-axis dialog box



Parameter		Description
Display mode		Select the display mode.
	Auto	If this option is activated, the Y-axis is automatically scaled according to the captured values. No further entries have to be set.
	Fixed	If this option is activated, the displayed section of the Y-axis has to be defined by the minimum and maximum value.
Minimum		This value defines the minimum displayed value of the Y-axis.
Maximum		This value defines the maximum displayed value of the Y-axis.
Grid		If this option is activated, a grid will be displayed. Select the color of the grid lines from the list.
Description		If this option is activated, the Y-axis is labeled with the text entered in the field next to it.
Tick marks		–
	Fixed spacing	Select this check box to scale the axis in certain distances.
	Distance	Enter a positive distance. Default: 1
	Subdivision	Enter a number of subdivisions for each distance.
Font		Opens the standard dialog box for defining the font for the trace display.

Section 24.3

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 SoMachine, 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 SoMachine, 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 (*see page 489*).

Section 24.4

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 <i>SoMachine, Menu Commands, Online Help</i>).
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 <i>SoMachine, Menu Commands, Online Help</i>).
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 <i>SoMachine, Menu Commands, Online Help</i>).

Actions	By Keyboard Operation	By Mouse Operation
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.

Chapter 25

Symbol Configuration Editor

What Is in This Chapter?

This chapter contains the following topics:

Topic	Page
Symbol Configuration Editor	492
Symbol Configuration	495
Adding a Symbol Configuration	496

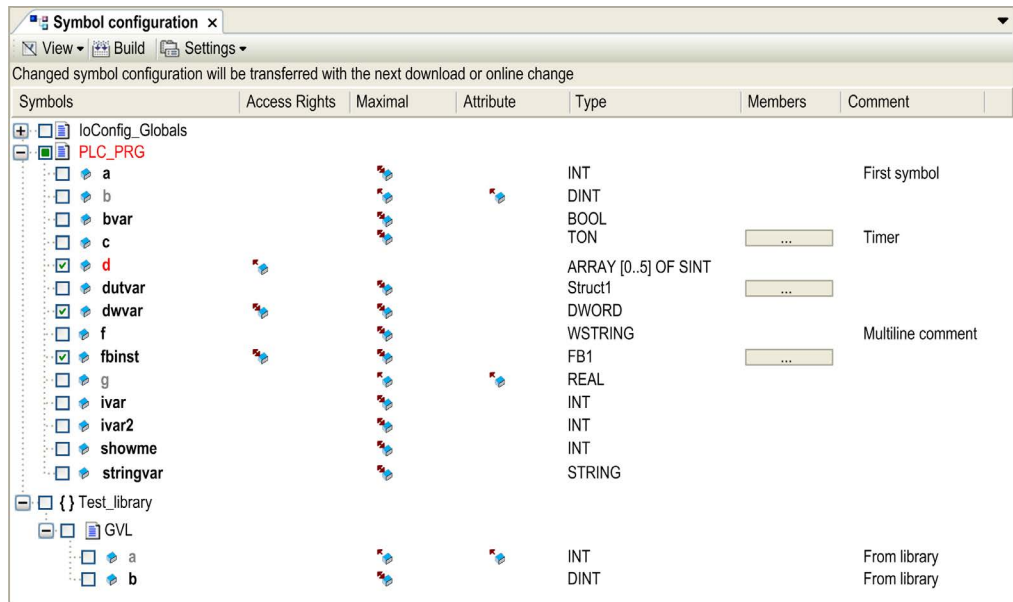
Symbol Configuration Editor

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 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.

Symbol configuration editor



Symbols	Access Rights	Maximal	Attribute	Type	Members	Comment
IoConfig_Globals						
PLC_PRG						
a				INT		First symbol
b				DINT		
bvar				BOOL		
c				TON	...	Timer
d				ARRAY [0..5] OF SINT	...	
dutvar				Struct1	...	
dwvar				DWORD		
f				WSTRING		Multiline comment
fbinst				FB1	...	
g				REAL		
ivar				INT		
ivar2				INT		
showme				INT		
stringvar				STRING		
Test_library						
GVL						
a				INT		From library
b				DINT		From library

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 POU or libraries are listed in the **Symbols** column. You can expand them in order to show the particular variables.

Description of the Elements

The **View** button that is available in the toolbar above the table allows you to set the following filters to reduce the number of displayed variables:

Filter	Description
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 when only the already configured variables are displayed (see the 2 filters described above). It has the effect that also those variables will be 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 currently set for the variable by the pragma. Refer to the following description of access right.

To modify the access rights for a selected item, click the symbol in the **Access Rights** column. Each mouse-click will switch the symbol within the following definitions: read+write , write-only , read-only .

The **Maximal** column shows which right maximally can be set.

The **Comment** column shows any comments which have been added in the declaration of the variable.

With the POU property **Link Always**, an uncompiled object can be reinterpreted and downloaded to the controller. If this property is set in the **Build** tab of the **Properties** dialog box of the selected object, then all variables declared in this object will be available, even if the object itself is not referenced by other code. In addition, you can use the pragma {attribute linkalways} (*see page 576*) to make not compiled variables available in the symbol configuration.

Variables which are configured to be exported but currently are not valid in the application - for example because their declaration has been removed - will be shown in red. This also applies to the concerned POU or library name.

The column **Type** also shows alias data types, as for example, MY_INT : INT. For a variable declared with data type MY_INT, whereby MY_INT is declared as follows:
 TYPE MY_INT : INT; END_TYPE.

To get symbols for a variable of a structured data type, like for other variables first of all activate the item in the **Symbols** column. This basically will have the effect that for all members of the structure, symbols will be exported to the symbol file. This can result in a large number of entries in the symbol file, which not at all are needed. You may also select only particular member variables. You can do this in the dialog box **Symbol Configuration for Data Type**. Click the ... button in the **Members** column to open this dialog box. In case of nested types, this dialog box will again provide a button to open another data type symbol configuration dialog box.

The editor view is automatically refreshed at a build run. The toolbar provides a **Build** button for quick access.

The toolbar button **Settings** allows you to activate the option **Include comments in XML**. This has the effect that comments assigned to variables will also be exported to the symbol file.

NOTE: Do not activate the option **Include comments in XML** for projects with Vijeo-Designer. Otherwise, Vijeo-Designer will not operate properly and may cease to function entirely.

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.

In case of using a device which supports a separate application file for the symbol configuration (also refer to the Symbol Configuration chapter ([see page 495](#))), a **Download** button will be available in the toolbar. You can use it to initiate an immediate new download of the *<application name>._Symbols* file in case the symbol configuration has been modified in online mode.

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. The description of the symbols will be available in an XML file (symbol file) in the project directory. It will be downloaded to the controller together with the application.

Symbol Information

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*

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 ([see page 119](#)).

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 ([see page 493](#)).

For example, concerning the maximum number of applications on a controller, the symbol application has to be handled as a normal application.

Adding a Symbol Configuration

Prerequisites

Variables that will be exchanged between the controller and (multiple) HMI devices using the transparent SoMachine protocol (*see SoMachine, Introduction*) must be published in the controller using the **Symbol configuration**. They will then be available as SoMachine variables in Vijeo-Designer.

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 will automatically include the IECVarAccess.library in the **Library Manager**.

You can define the variables to be exported as symbols in the symbol configuration editor (*see page 492*) or via pragmas (attribute symbol (*see page 594*)), 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.

Another possibility is provided by the SFC editor: You can define the implicitly created element flags in the element properties (*see 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, enter a Name for the symbol configuration in the text box.
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 1 symbol configuration node can be created per device.

For details on the variables interchange between the controller and HMI part, refer to the chapter SoMachine Controller-HMI Data Exchange ([see page 499](#)).

Chapter 26

SoMachine Controller - HMI Data Exchange

What Is in This Chapter?

This chapter contains the following topics:

Topic	Page
SoMachine Single Variable Definition	500
Publishing Variables in the Controller Part	504
Selecting Variables in the HMI Part	506
Publishing Variables in the HMI Part	507
Parametrization of the Physical Media	509
Communication Performance on Controller - HMI Data Exchange	510

SoMachine Single Variable Definition

Overview

By publishing the variable(s) in SoMachine, they will automatically be available for use in the Vijeo-Designer HMI application.

For variable exchange with the SoMachine protocol, perform the following steps:

- Create variables in the controller part.
- Publish the variables by defining them as **Symbols** in the controller part. They are now available in the HMI part as SoMachine variables.
- Configure the physical connection (automatically setup by SoMachine).

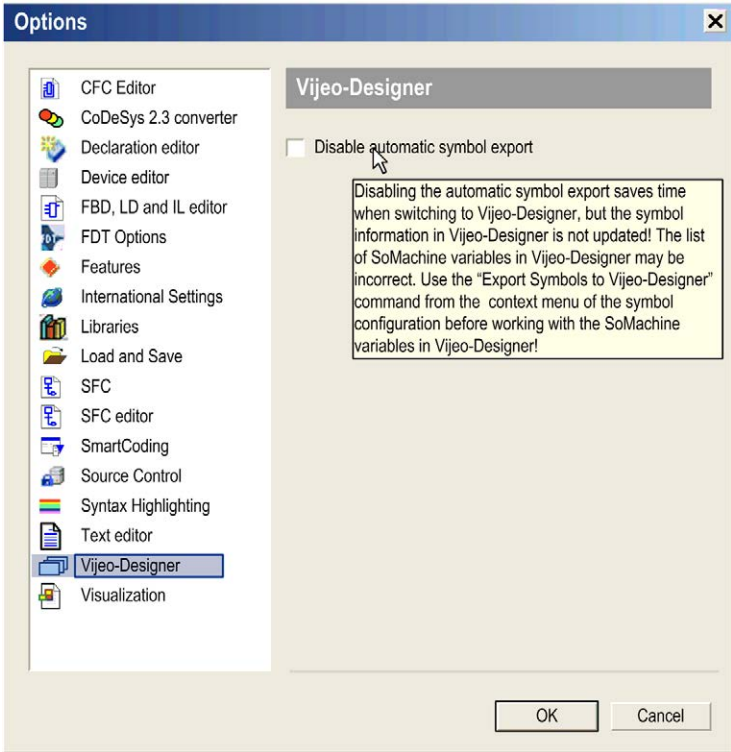
NOTE: The last step is not necessary for XBTGC controllers because they may communicate with their own control variables.

Disabling Automatic Symbol Export to Vijeo-Designer

By default, SoMachine 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 SoMachine 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 will be displayed.
2	Select the entry Vijeo-Designer from the list on the left-hand side.

Step	Action
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 SoMachine 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.

Variable Types for SoMachine - HMI Data Exchange

The following table lists the variable types for SoMachine - HMI data exchange:

Variable Type in SoMachine	Variable Type in Vijeo-Designer	Comment
BOOL	BOOL	---
BYTE	Integer	---
WORD	UINT	---
DWORD	UDINT	---
SINT	Integer	---
INT	INT	---
DINT	DINT	---
USINT	Integer	---
UINT	UINT	---
UDINT	UDINT	---
REAL	REAL	---
STRING	STRING	---
WSTRING	STRING	WSTRING is supported in Vijeo-Designer as a general STRING type. This means that you can either exchange only STRINGS or only WSTRINGS with the HMI. A mixture of these 2 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.
Array	–	In Vijeo-Designer, you can only reference the elements of an array, not the whole array. Example: Your array consists of SINTs called myVa lues. In Vijeo-Designer, you can reference myVa lues[0] or myVa lues[5] and put this into a variable on the HMI controller. 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 will be issued.
DUT	–	In Vijeo-Designer you can only reference the elements of a DUT, not the whole DUT. This behavior is similar to the behavior of arrays.

Unsupported Variable Types

The following variable types are not supported for SoMachine - HMI data exchange:

- all 64-bit integer formats
- LREAL
- all time and date formats
- non-zero based arrays: you cannot import an array that is defined, example:
`myArray[1..100]`.
- arrays of arrays: you cannot import an array that has an array as its element type, such as
`ARRAY [0..9] OF ARRAY [0..9] OF INT`. Nevertheless, you can use multi-dimensional arrays, such as `ARRAY [0..9, 0..9] OF INT`.

NOTE: The variables from the PLC_R structures of the PLCSystem library cannot be shared via the **Symbol Configuration** with the Vijeo-Designer application of HMI targets (including HMI controllers).

NOTE: Do not share references to a structured variable in the symbol configuration editor, as their values will not be displayed correctly on the HMI.

For further information on variable types for SoMachine - HMI data exchange, see the Vijeo-Designer online help.

Identifier Length

In Vijeo-Designer, the maximum length of the Symbol name is limited to 32 characters.

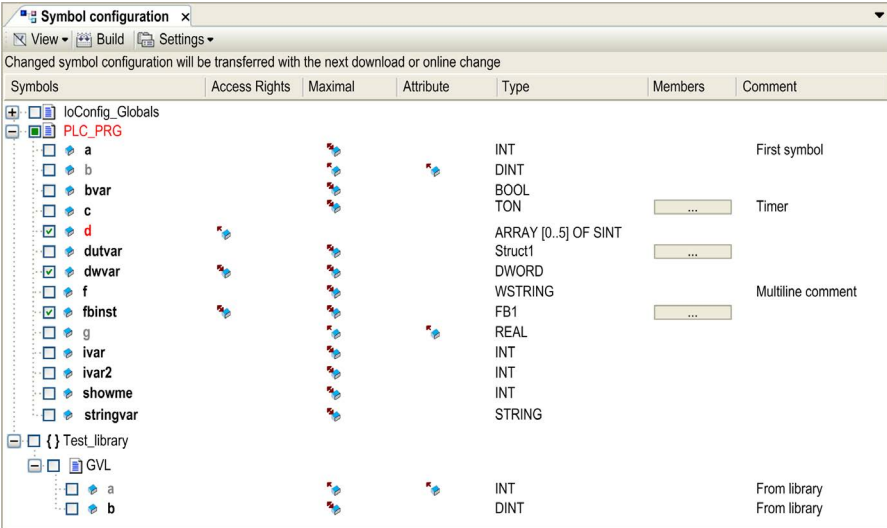
Publishing Variables in the Controller Part

Overview

Publish variables in the controller part of the SoMachine application within the **Symbol configuration** editor or in the **Variables** view of the software catalog (*see page 34*) of a POU.

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 <i>Adding a Symbol Configuration</i> chapter (<i>see page 496</i>).
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 (<i>see page 492</i>). Note: Only variables on elementary data types 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.

Publishing Variables in the Variables View of the Software Catalog

To publish variables in the **Variables** view of the software catalog (*see page 34*), proceed as follows:

Step	Action
1	Open the Variables view of the software catalog (<i>see page 34</i>).
2	To publish a variable, select the respective check box in the Publish column.

Scope	Name	Data type	Ad...	Init...	Comment	Attr...	Publish
MyController	GVL_ATV32_Node01						
	VAR_GLOBAL xMCB_rdy	BOOL			Motor Circuit B...		☒
	VAR_GLOBAL lActVelo	INT			Indicates the a...		☒
	VAR_GLOBAL xCmdEnPwr	BOOL			Command to p...		☒
	VAR_GLOBAL xCmdRst	BOOL			Command ackn...		☒
	VAR_GLOBAL xCmdStop	BOOL			Command to st...		☒
	VAR_GLOBAL xCmdJogFwd	BOOL			Comm... to tu...		☒
	VAR_GLOBAL xCmdJogRev	BOOL			Comm... to tu...		☒
	VAR_GLOBAL iSetJogVelo	INT			Setpoint "jog v...		☒
	VAR_GLOBAL xCmdMovVelo	BOOL			Command to st...		☒
	VAR_GLOBAL iSetMovVelo	INT			Setpoint "move...		☒
	VAR_GLOBAL xStatEnbl	BOOL			Indicates the A...		☒
	VAR_GLOBAL wErrID	WORD			Provides the er...		☒
	VAR_GLOBAL xErr	BOOL			Indicates an er...		☒
	VAR_GLOBAL xVeloActv	BOOL			If the drive is o...		☒
	VAR_GLOBAL xJogActv	BOOL			If the drive is o...		☒
	VAR_GLOBAL xComOk	BOOL			Indicates the C...		☒
	VAR_GLOBAL eComSta	CIA405.DE...			Node state pro...		☒
	GVL_ATV32_Node04						
	VAR_GLOBAL xMCB_rdy	BOOL			Motor Circuit B...		☒
	VAR_GLOBAL lActVelo	INT			Indicates the a...		☒
	VAR_GLOBAL xCmdEnPwr	BOOL			Command to p...		☒
	VAR_GLOBAL xCmdRst	BOOL			Command ackn...		☒
	VAR_GLOBAL xCmdStop	BOOL			Command to st...		☒
	VAR_GLOBAL xCmdJogFwd	BOOL			Comm... to tu...		☒
	VAR_GLOBAL xCmdJogRev	BOOL			Comm... to tu...		☒
	VAR_GLOBAL iSetJogVelo	INT			Setpoint "jog v...		☒
	VAR_GLOBAL xCmdMovVelo	BOOL			Command to st...		☒
	VAR_GLOBAL iSetMovVelo	INT			Setpoint "move...		☒

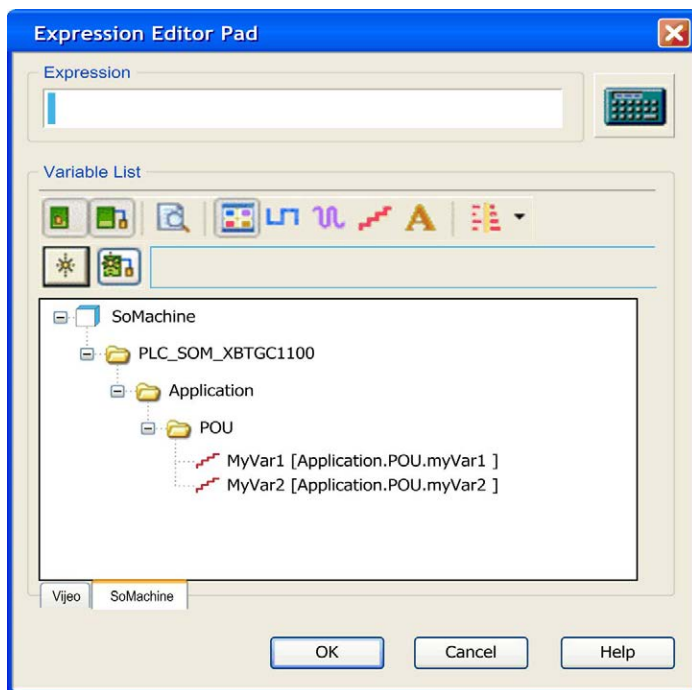
NOTE: Verify that the POU of the selected variables is called in a task. Otherwise the selected variables will not be published.

Selecting Variables in the HMI Part

Selecting Variables

Those variables that have been published in the controller part are directly available in the HMI part.

In the **Expression Editor Pad** of Vijeo-Designer, select the **SoMachine** tab to have direct access to the variables published in SoMachine.



For further information, refer to the Vijeo-Designer online help.

Publishing Variables in the HMI Part

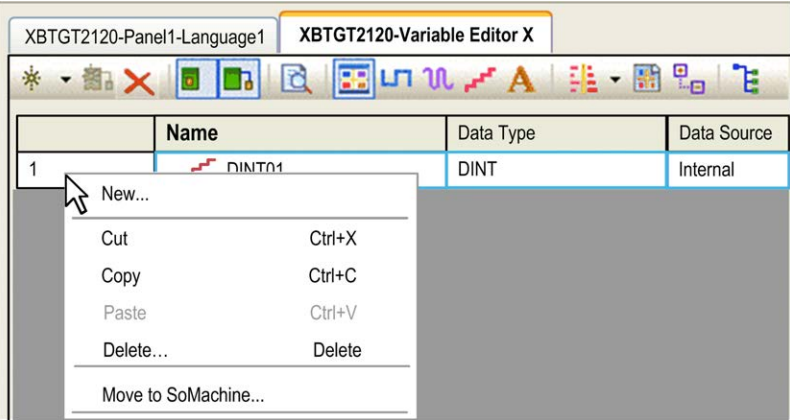
Supported Variable Types

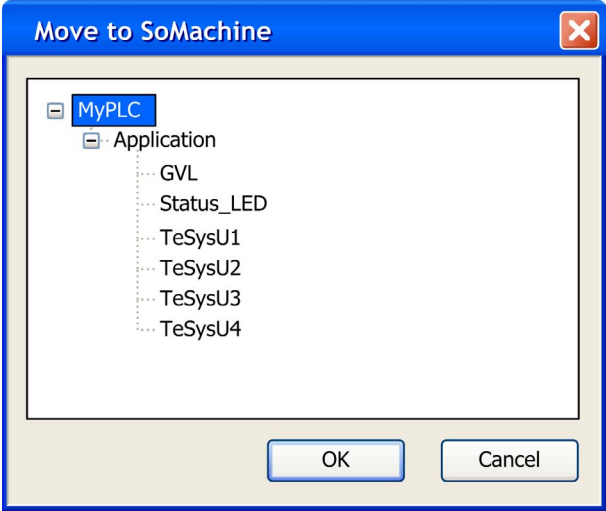
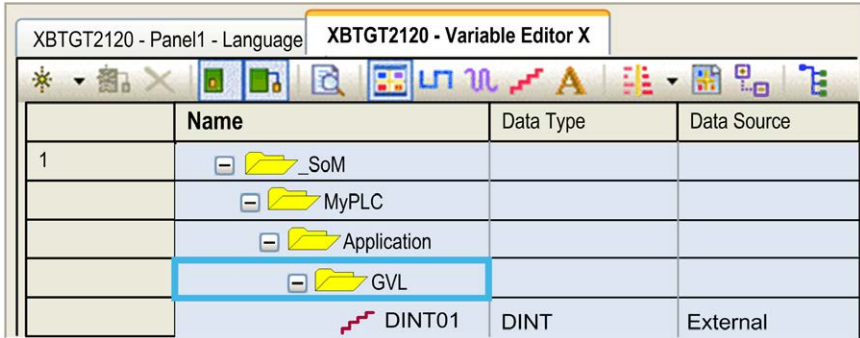
The following variable types can be published in Vijeo-Designer to make them available for the entire SoMachine project:

- 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 context menu. The screenshot shows the 'XBTGT2120-Variable Editor X' window. It contains a table with columns 'Name', 'Data Type', and 'Data Source'. The first row has 'DINT01' under 'Name', 'DINT' under 'Data Type', and 'Internal' under 'Data Source'. A right-click context menu is open over the 'DINT01' cell, showing options: 'New...', 'Cut' (Ctrl+X), 'Copy' (Ctrl+C), 'Paste' (Ctrl+V), 'Delete...', 'Delete', and 'Move to SoMachine...'. The 'Move to SoMachine...' option is highlighted at the bottom of the menu.
	Result: The Move to SoMachine dialog box will be displayed.
3	In the Move to SoMachine dialog box open the subfolders of the devices defined in SoMachine to see the levels where variables are defined (POU or GVL).

Step	Action
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 SoMachine POU or GVL and is / are available throughout the SoMachine project. 

Parametrization 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.

Configuration Example

The default settings below are valid for communications between M238 and an HMI panel via serial line RS485 using an XBTZ9008 cable (serial line SubD-RJ45).

Configuration of M238 with HMI panel:

M238 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: SoMachine - network with at least 1 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 dialog)

Configuration of XBTGC:

Device	Configuration
XBTGC controller	no configuration required
HMI panel	IO-Manager
Driver	SoMachine - Combo with at least 1 Scan-Group

Communication Performance on Controller - HMI Data Exchange

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 SoMachine 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

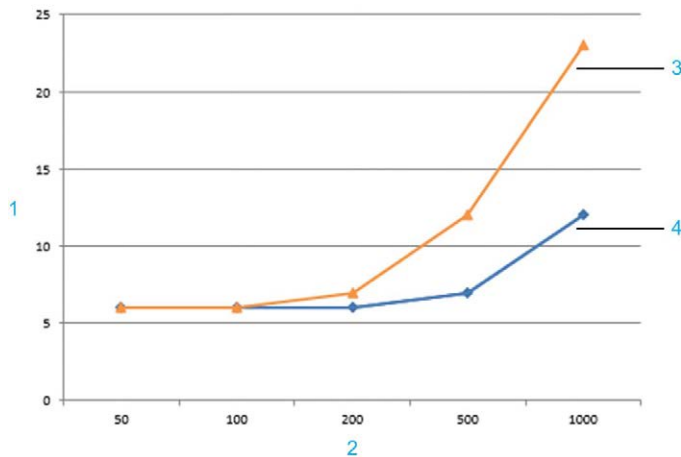
To improve the communication performance, you can take the following measures:

- 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 Displaying Values After Establishing the Controller-HMI-Connection

The graph indicates reference values that have been measured for the time that is required to display the values on the HMI panels when establishing a connection over the SoMachine protocol via Ethernet (for example, after downloading applications). The reference values are typically representative of the performance of the XBTGT HMI Controllers or M258 Logic Controllers. The reference values were obtained by using different numbers of variables under non-industrial conditions.

Typical delay to establish a connection and display values on the HMI panel:

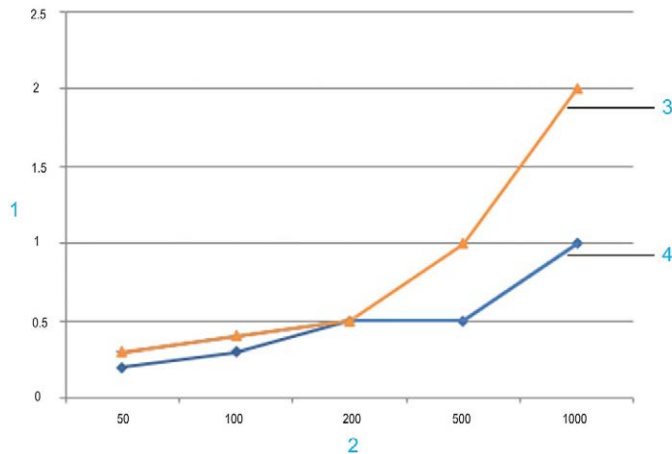


- 1 Time in seconds
- 2 Number of variables
- 3 XBTGT2330 + M258
- 4 XBTGT4330 + M258

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 SoMachine protocol via Ethernet between XBTGT HMI and M258 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 + M258
- 4 XBTGT4330 + M258

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 SoMachine **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.

Part VII

Programming Reference

What Is in This Part?

This part contains the following chapters:

Chapter	Chapter Name	Page
27	Variables Declaration	517
28	Data Types	599
29	Programming Guidelines	625
30	Operators	639
31	Operands	731

Chapter 27

Variables Declaration

What Is in This Chapter?

This chapter contains the following sections:

Section	Topic	Page
27.1	Declaration	518
27.2	Variable Types	535
27.3	Method Types	545
27.4	Pragma Instructions	550
27.5	Attribute Pragmas	562
27.6	The Smart Coding Functionality	596

Section 27.1

Declaration

What Is in This Section?

This section contains the following topics:

Topic	Page
General Information	519
Recommendations on the Naming of Identifiers	522
Variables Initialization	527
Declaration	528
Shortcut Mode	529
AT Declaration	530
Keywords	531

General Information

Overview

You can declare variables:

- in the **Variables** view of the **Software Catalog** (*see page 33*)
- in the **Declaration Editor** of a POU (*see page 382*)
- via the **Auto Declare** dialog box (*see page 528*)
- in a DUT editor
- in a GVL editor

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 (*see page 382*), 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_EXTERNAL
- VAR_CONFIG

The variable type keywords may be supplemented by attribute keywords (*see page 539*).

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 2 different identifiers.
Do not use more than 1 underscore character in a row.
- unlimited length
- recommendations concerning multiple use (see next paragraph)

Also, consider the recommendations given in chapter Recommendations on the Naming of Identifiers ([see page 522](#)).

Multiple Use of Identifiers (Namespaces)

The following outlines the regulations concerning the multiple use of identifiers:

- Do not create an identifier that is identical to a keyword.
- 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` in library `lib1` 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`.

AT <address>

You can link the variable directly to a definite address (*see page 530*) 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 (*see page 600*), optionally extended by an :=< initialization> (*see page 527*).

Pragma Instructions

Optionally, you can add pragma instructions (*see page 550*) in the declaration part of an object in order to affect the code generation for various purposes.

Hints

Automatic declaration (*see page 528*) of variables is also possible.

For faster input of the declarations, use the shortcut mode (*see page 529*).

Recommendations on 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 POU's (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 (see page 519)*), consider the following recommendations in order to make the naming as unique as possible:

- Variable names (see page 522)
- Variable names in Libraries (see page 524)
- User-defined data types (DUTs) in Libraries (see page 525)
- Functions, Function blocks, Programs (POU), Actions (see page 525)
- POU's in Libraries (see page 526)
- Visualization names (see page 526)

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	not for arithmetic operations
SINT	–128	127	8 bit	si	–
* 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).					

Data Type	Lower Limit	Upper Limit	Information Content	Prefix	Comment
USINT	0	255	8 bit	usi	–
INT	–32,768	32,767	16 bit	i	–
UINT	0	65,535	16 bit	ui	–
DINT	–2,147,483,648	2,147,483,647	32 bit	di	–
UDINT	0	4,294,967,295	32 bit	udi	–
LINT	-2^{63}	$2^{63}-1$	64 bit	li	–
ULINT	0	$2^{64}-1$	64 bit	uli	–
REAL	–	–	32 bit	r	–
LREAL	–	–	64 bit	lr	–
STRING	–	–	–	s	–
TIME	–	–	–	tim	–
TIME_OF_DAY	–	–	–	tod	–
DATE_AND_TIME	–	–	–	dt	–
DATE	–	–	–	date	–
ENUM	–	–	16 bit	e	–
POINTER	–	–	–	p	–
ARRAY	–	–	–	a	–

* 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: sdo).

Example

```
cansdoReceivedTelegram: CAN_SDOTelegram;  
TYPE CAN_SDOTelegram :      (* prefix: sdo *)  
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_ and global constants are prefixed by 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 a short expressive description (for example, SDOTe legram) of the structure.

Example (in library with namespace CAL):

```
TYPE Day : (
  MONDAY,
  TUESDAY,
  WEDNESDAY,
  THURSDAY,
  FRIDAY,
  SATURDAY,
  SUNDAY);
```

Declaration:

```
eToday : CAL.Day;
```

Use in application:

```
IF eToday = CAL.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 are prefixed by an expressive short name of the POU (for example, SendTe legram). As with variables, the first letter of a word of the POU name should always be a capital letter whereas the others should be small letters. It is recommended to compose the name of the POU of a verb and a substantive.

Example

```
FUNCTION_BLOCK SendTelegram (* prefix: canst *)
```

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_.

Each function - for the reason of compatibility with previous software versions - is supposed to have at least one parameter. Do not use structures as return values in external functions.

POUs in Libraries

Structure

For creating method names, the same rules apply as for actions. Enter English comments for possible inputs of a method. Add a short description of a method to its declaration. Start interface names with letter I; for example, ICANDevice.

NOTE: Consider the usage of the namespace when using POUs declared in libraries.

Visualization Names

Avoid naming a visualization similar to another object in the project because this would cause anomalies in case of visualization changes.

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 IEC6113
1-3:
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 ([see page 614](#))
- initialization of structures ([see page 617](#))
- initialization of a variable with a subrange type ([see page 621](#))

NOTE: Variables of global variables lists (GVL) are initialized before local variables of a POU.

Declaration

Declaration Types

You can declare variables manually by using the textual or tabular declaration editor (*see page 382*) 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 (*see page 519*).

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, you can edit the declaration of this variable.

Shortcut Mode

Overview

The declaration editor (*see page 382*) 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	ST S 2; A string	ST:STRING(2); (* A string *)
4	X %MD12 R 5 Real Number	X AT %MD12: REAL := 5.0; (* 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

<identifier> AT <address> : <data type>;

A valid address has to follow the keyword AT. For further information, refer to the *Address* description ([see page 747](#)). 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 POU's.
- 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

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.

Explanation: booleans themselves are actually 8 bits when declared, and so when they are written to other variable types, all 8 bits go along.

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 POU's:

- ABS
- ACOS
- ACTION (only used in the export format)
- ADD
- ADR
- AND
- ANDN
- ARRAY
- ASIN
- AT
- ATAN
- BITADR
- BOOL
- BY
- BYTE
- CAL
- CALC
- CALCN
- CASE
- CONSTANT
- COS
- DATE
- DINT
- DIV
- DO
- DT
- DWORD
- ELSE
- ELSIF
- END_ACTION (only used in the export format)
- END_CASE
- END_FOR
- END_FUNCTION (only used in the export format)
- END_FUNCTION_BLOCK (only used in the export format)
- END_IF
- END_PROGRAM (only used in the export format)
- END_REPEAT
- END_STRUCT

- END_TYPE
- END_VAR
- END_WHILE
- EQ
- EXIT
- EXP
- EXPT
- FALSE
- FOR
- FUNCTION
- FUNCTION_BLOCK
- GE
- GT
- IF
- INDEXOF
- INT
- JMP
- JMPC
- JMPCN
- LD
- LDN
- LE
- LINT
- LN
- LOG
- LREAL
- LT
- LTIME
- LWORD
- MAX
- METHOD
- MIN
- MOD
- MOVE
- MUL
- MUX
- NE
- NOT
- OF
- OR
- ORN
- PARAMS
- PERSISTENT
- POINTER
- PROGRAM

- R
- READ_ONLY
- READ_WRITE
- REAL
- REFERENCE
- REPEAT
- RET
- RETAIN
- RETC
- RETCN
- RETURN
- ROL
- ROR
- S
- SEL
- SHL
- SHR
- SIN
- SINT
- SIZEOF
- SUPER
- SQRT
- ST
- STN
- STRING
- STRUCT
- SUPER
- SUB
- TAN
- THEN
- THIS
- TIME
- TO
- TOD
- TRUE
- TRUNC
- TYPE
- UDINT
- UINT
- ULINT
- UNTIL
- USINT
- VAR
- VAR_ACCESS (only used specifically, depending on the hardware)
- VAR_CONFIG

- VAR_EXTERNAL
- VAR_GLOBAL
- VAR_IN_OUT
- VAR_INPUT
- VAR_OUTPUT
- VAR_STAT
- VAR_TEMP
- WHILE
- WORD
- WSTRING
- XOR
- XORN

Additionally, the conversion operators as listed in the **Input Assistant** are handled as keywords.

Section 27.2

Variable Types

What Is in This Section?

This section contains the following topics:

Topic	Page
Variable Types	536
Attribute Keywords for Variable Types	539
Variables Configuration - VAR_CONFIG	543

Variable Types

Overview

This chapter provides further information on the following variable types:

- VAR local variables ([see page 536](#))
- VAR_INPUT input variables ([see page 536](#))
- VAR_OUTPUT output variables ([see page 536](#))
- VAR_IN_OUT input and output variables ([see page 537](#))
- VAR_GLOBAL global variables ([see page 537](#))
- VAR_TEMP temporary variables ([see page 538](#))
- VAR_STAT static variables ([see page 538](#))
- VAR_EXTERNAL external variables ([see page 538](#))

Local Variables - VAR

Between the keywords VAR and END_VAR, all local variables of a POU are declared ([see page 519](#)). These have no external connection; in other words, they cannot be written from the outside.

Consider the possibility of adding an attribute ([see page 539](#)) 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 ([see page 519](#)) that serve as input variables for a POU. This means that at the call position, the value of the variables can be provided along with a call.

Consider the possibility of adding an attribute ([see page 539](#)).

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. This means that these values are carried back to the POU that makes the call.

Consider the possibility of adding an attribute ([see page 539](#)) 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 (*see page 519*) 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). This means that 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 via <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 SoMachine, Menu Commands, Online Help*).

Example

```
VAR_IN_OUT
  iInOut1:INT; (* 1. Inputoutputvariable *)
END_VAR
```

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 (*see page 539*) 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* (*see page 727*).

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 at the first call of the respective POU.

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 (*see page 539*) 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

```
VAR_EXTERNAL
  iVarExt1:INT; (* 1st external variable *)
END_VAR
```

Attribute Keywords for Variable Types

Overview

You can add the following attribute keywords to the declaration ([see page 519](#)) of the variable type in order to specify the scope:

- RETAIN: refer to *Retain Variables* ([see page 539](#))
- PERSISTENT: refer to *Persistent Variables* ([see page 540](#))
- CONSTANT: refer to *Constants - CONSTANT* ([see page 541](#)), *Typed Literals* ([see page 541](#))

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* ([see page 540](#))).

NOTE: A VAR PERSISTENT declaration is interpreted in the same way as a VAR PERSISTENT RETAIN or VAR RETAIN PERSISTENT.

NOTE: Use the command ([see SoMachine, Menu Commands, Online Help](#)) **Add all instance paths** to take variables declared as persistent into the **Persistent list** object.

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
```

Using interfaces or function blocks out of System Configuration libraries 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.

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.

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 ([see page 541](#)).

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.

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 POU's.)

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 name. In case of inconsistencies, you will be prompted to reinitialize all persistent variables (*see SoMachine, 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.

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 or globally.

Syntax

VAR CONSTANT<identifier>:<type> := <initialization>;END_VAR

Example

```
VAR CONSTANT
  c_iCon1:INT:=12; (* 1. Constant*)
END_VAR
```

Refer to the *Operands* chapter (*see page 731*) for a list of possible constants.

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.

Variables Configuration - VAR_CONFIG

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 ([see page 747](#)) 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 POU's 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 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.

Section 27.3

Method Types

What Is in This Section?

This section contains the following topics:

Topic	Page
FB_init, FB_reinit Methods	546
FB_exit Method	549

FB_init, FB_reinit Methods

FB_Init

The FB_init method serves to reinitialize a function block or a structure. FB_init can be declared explicitly for function blocks, but in any case is always available implicitly.

The FB_init method contains initialization code for the function block or structure as declared in the declaration part of the respective object. If the init method is declared explicitly, the implicit initialization code will be inserted into this method. The programmer can add further initialization code.

NOTE: When execution reaches the user-defined initialization code, the function block, respectively the structure, is already fully initialized via the implicit initialization code.

The init method is called after download for each declared function block instance and each variable of a structure type.

NOTE: Executing the **Online Change** command will overwrite the initialization values by previous values.

For information on the call sequence in case of inheritance, refer to the respective paragraph in the *FB_exit Method* chapter ([see page 549](#)).

For information on the possibility of defining a function block instance method to be called automatically after initialization via FB_init, refer to chapter *Attribute call_after_init* ([see page 566](#)).

Interface of the init method

```
METHOD fb_init : BOOL
VAR_INPUT
    bInitRetains : BOOL; // if TRUE, the retain variables are initialized
    (warm start / cold start)
    bInCopyCode : BOOL; // if TRUE, the instance afterwards
    gets moved into the copy code (online change)
END_VAR
```

The return value is inconsequential.

NOTE: Consider also the possible use of an exit method and the resulting execution order (refer to chapter *FB_exit Method* ([see page 549](#))).

User-defined input

In an init method, you can declare additional function block inputs. Assign them at the declaration of a function block instance.

Example for an init method for a function block serialdevice:

```

METHOD PUBLIC FB_init : bool
VAR_INPUT
    bInitRetains : BOOL; // Initialization of the retain variables
    bInCopyCode : BOOL; // Instance moved into copy code
    nCOMnum : INT; // Input: number of the COM interface to listen at
END_VAR

```

Declaration of function block serialdevice:

```

    com1: serialdevice (nCOMnum:=1);
    com0: serialdevice (nCOMnum:=0);

```

Example for using FB_init for a structure DUTxy:

Structure DUTxy

```

TYPE DUTxy :
STRUCT
    a: INT := 10;
    b: INT := 11;
    c: INT := 12;
END_STRUCT
END_TYPE

```

Calling fb_init for reinitialization:

```

PROGRAM PLC_PRG
VAR
    dutTest: DUTxy;
    xInit: BOOL:=FALSE;
END_VAR
IF xinit THEN // if xinit is set TRUE, then the
reinitialization via fb_init to the values as defined in DUTxy will be
done
    dutTest.FB_Init(TRUE,TRUE);
ELSE
    dutTest.a := 1;
    dutTest.b := 2;
    dutTest.c := 3;
END_IF

```

FB_reinit

If a method named `FB_reinit` is declared for a function block instance, this method will be called when the instance is copied. This is the case at an online change after a modification of the function block declaration. The method will cause a reinitialization of the new instance module that has been created by the copy code. A reinitialization can be desired because the data of the original instance will be written to the new instance after the copying, even though you want to obtain the original initialization values. The `FB_reinit` method has no inputs. Consider that in contrast to `FB_init` the method has to be declared explicitly. If a reinitialization of the basic function block implementation is desired, `FB_reinit` has to be called explicitly for that POU.

The `FB_reinit` method has no inputs.

For information on the call sequence in case of inheritance, refer to the respective paragraph in the *FB_exit Method* chapter ([see page 549](#)).

FB_exit Method

Overview

The `FB_exit` method is a special method for a function block. It has to be declared explicitly as there is no implicit declaration. The exit method, if present, is called for all declared instances of the function block before a new download, at a reset or during online change for all moved or deleted instances.

Interface of the `FB_exit` method

There is only one mandatory parameter:

```
METHOD fb_exit : BOOL
VAR_INPUT
    bInCopyCode : BOOL; // if TRUE, the exit method is called
    for exiting an instance that is copied afterwards (online change).
END_VAR
```

Consider also the *FB_init method* ([see page 546](#)) and the following execution order:

1	exit method: exit old instance	<code>old_inst.fb_exit(bInCopyCode := TRUE);</code>
2	init method: initialize new instance	<code>new_inst.fb_init(bInitRetains := FALSE, bInCopyCode := TRUE);</code>
3	copy function block values (copy code)	<code>copy_fub(&old_inst, &new_inst);</code>

Inheritance

Besides this, in case of inheritance, the following call sequence is TRUE (assuming or the POU's used for this listing: `SubFB EXTENDS MainFB` and `SubSubFB EXTENDS SubFB`):

```
fbSubSubFb.FB_Exit(...);
fbSubFb.FB_Exit(...);
fbMainFb.FB_Exit(...);
fbMainFb.FB_Init(...);
fbSubFb.FB_Init(...);
fbSubSubFb.FB_Init(...);
```

For `FB_reinit`, the following applies:

```
fbMainFb.FB_reinit(...);
fbSubFb.FB_reinit(...);
fbSubSubFb.FB_Init(...);
```

Section 27.4

Pragma Instructions

What Is in This Section?

This section contains the following topics:

Topic	Page
Pragma Instructions	551
Message Pragmas	553
Conditional Pragmas	555

Pragma Instructions

Overview

A pragma instruction is used to affect the properties of 1 or several variables concerning the compilation or precompilation (preprocessor) process. This means that 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.

A pragma can determine whether a variable will be initialized, monitored, added to a parameter list, added to the symbol list (*see page 594*), 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.

Further Information

Depending on the type and contents of a pragma, the pragma operates on the subsequent statement, respectively all subsequent statements, until 1 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 Pragas* ([see page 553](#))
- *Attribute Obsolete* ([see page 590](#))
- *Attribute Pragas* ([see page 563](#))
- *Conditional Pragas* ([see page 555](#))
- *Attribute Symbol* ([see page 594](#))

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* (*see page 555*).

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 (<i>see page 590</i>), this alert is explicitly defined for the local position.
{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:

Messages			
Build			
Description	Project	Object	Position
----- Build started: Application: Res.App2 -----			
typify code...			
❗ Compile time before typification: 0 ms			
❗ Compile time after typification: 15 ms			
❗ TODO: should get another name!	TS pragma	NewPOU	Line 3 (Decl)
⚠ This is an alert	TS pragma	NewPOU	Line 7 (Impl)
Part xy has been compiled completely	TS pragma	NewPOU	Line 10 (Impl)
Compile complete -- 1 errors, 2 warnings			

Conditional Pragmas

Overview

The ExST (Extended ST) language supports several conditional *Pragma Instructions* ([see page 551](#)), which affect the code generation in the precompile or compile process.

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.

<code>{define identifier string}</code>	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 <code>{undefine}</code> directive. Used for conditional compilation (see page 556).
<code>{undefine identifier}</code>	The preprocessor definition of the identifier (by <code>{define}</code> , 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.
<code>{IF expr}</code> ... <code>{ELIF expr}</code> ... <code>{ELSE}</code> ... <code>{END_IF}</code>	These are pragmas for conditional compilation. The specified expressions <code>expr</code> s 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 <code>elsif</code> and <code>else</code> sections are optional, and <code>elsif</code> sections may appear arbitrarily more often. Within the constant <code>expr</code> , you can use several conditional compilation operators (see page 556).

Conditional Compilation Operators

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. Consider that 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**).

The following operators are supported:

- `defined (identifier)` ([see page 556](#))
- `defined (variable:variable)` ([see page 557](#))
- `defined (type:identifier)` ([see page 557](#))
- `defined (pou:pou-name)` ([see page 557](#))
- `hasattribute (pou: pou-name, attribute)` ([see page 558](#))
- `hasattribute (variable: variable, attribute)` ([see page 558](#))
- `hastype (variable:variable, type-spec)` ([see page 559](#))
- `hasvalue (define-ident, char-string)` ([see page 560](#))
- NOT operator ([see page 561](#))
- operator AND operator ([see page 561](#))
- operator OR operator ([see page 561](#))
- operator ([see page 561](#))

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. Variable `pdef1` is declared 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;
```

Additionally, an example for a message pragma ([see page 553](#)) 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)

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):

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 a POU or an action with that particular POU name is defined. Otherwise it is FALSE.

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}
```

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 POU's 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 the
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

- ANY
- ANY_DERIVED
- ANY_ELEMENTARY
- ANY_MAGNITUDE
- ANY_BIT
- ANY_STRING
- ANY_DATE
- ANY_NUM
- ANY_REAL
- ANY_INT
- LREAL
- REAL
- LINT
- DINT
- INT
- SINT
- ULINT
- UDINT
- UINT
- USINT

- TIME
- LWORD
- DWORD
- WORD
- BYTE
- BOOL
- STRING
- WSTRING
- DATE_AND_TIME
- DATE
- TIME_OF_DAY

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 var
iable test has value 1 *)
x := x + 1;
{ELSIF hasvalue(test, '2')}
(* the following code line will be processed in App1, because there var
iable test has value 2 *)
x := x + 2;
{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 A
pp2, because both contain at least one of the POU's "PLC_PRG1" and "Chec
kBounds" *)
bORTest := TRUE;
{END_IF}
```

(operator)

(operator) braces the operator.

Section 27.5

Attribute Pragmas

What Is in This Section?

This section contains the following topics:

Topic	Page
Attribute Pragmas	563
User-Defined Attributes	564
Attribute call_after_init	566
Attribute displaymode	567
Attribute ExpandFully	568
Attribute global_init_slot	569
Attribute hide	570
Attribute hide_all_locals	571
Attribute initialize_on_call	572
Attribute init_namespace	573
Attribute init_On_Onlchange	574
Attribute instance-path	575
Attribute linkalways	576
Attribute monitoring	578
Attribute namespace	582
Attribute no_check	584
Attribute no_copy	584
Attribute no-exit	585
Attribute no_init	586
Attribute no_virtual_actions	587
Attribute obsolete	590
Attribute pack_mode	591
Attribute qualified_only	592
Attribute reflection	593
Attribute subsequent	593
Attribute symbol	594
Attribute warning disable	595

Attribute Pragmas

Overview

You can assign attribute pragmas ([see page 551](#)) to a signature in order to influence the compilation or pre-compilation that is the code generation.

There are user-defined attributes ([see page 564](#)), which you can use in combination with conditional pragmas ([see page 555](#)).

There are also the following predefined standard attribute pragmas:

- attribute displaymode ([see page 567](#))
- attribute ExpandFully ([see page 568](#))
- attribute global_init_slot ([see page 569](#))
- attribute hide ([see page 570](#))
- attribute hide_all_locals ([see page 571](#))
- attribute initialize_on_call ([see page 572](#))
- attribute init_namespace ([see page 573](#))
- attribute init_On_Onlchange ([see page 574](#))
- attribute instance-path ([see page 575](#))
- attribute linkalways ([see page 576](#))
- attribute monitoring ([see page 578](#))
- attribute namespace ([see page 582](#))
- attribute no_check ([see page 584](#))
- attribute no_copy ([see page 584](#))
- attribute no-exit ([see page 585](#))
- attribute noint ([see page 586](#))
- attribute no_virtual_actions ([see page 587](#))
- attribute obsolete ([see page 590](#))
- attribute pack_mode ([see page 591](#))
- attribute qualified_only ([see page 592](#))
- attribute reflection ([see page 593](#))
- attribute subsequent ([see page 593](#))
- attribute symbol ([see page 594](#))
- attribute warning disable ([see page 595](#))

User-Defined Attributes

Overview

You can assign arbitrary user-defined or application-defined attribute pragmas to POU, type declarations, or variables. This attribute can be queried before compilation by conditional pragmas (*see page 555*).

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 POU 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* ([see page 555](#)).

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 this purpose, attach the attribute both to the function block itself and to the instance method to be called (for performance reasons). The method has to be called after `FB_Init` ([see page 546](#)) and after having applied the variable values of an initialization expression in the instance declaration.

This functionality is supported as from SoMachine V3.0.

Syntax

```
{attribute '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 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 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.

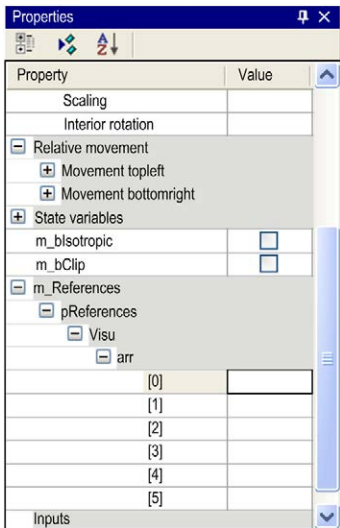
As input variable arr is defined in the interface editor of visu and will later be available for assignments in the **Properties** dialog box of the frame in visu_main.

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

You can apply the pragma `{attribute 'global_init_slot'}` only for signatures. Per default, the order of initializing the variables of a global variable list is arbitrary. However, in some cases, prescribing an order is necessary, that is if variables of 1 global list are referring to variables of another list. In this case, you can use the pragma to arrange the order for global initialization.

Syntax

```
{attribute 'global_init_slot' := '<value>'}
```

Replace the template `<value>` by an integer value describing the initialization order of the signature. The default value is 50,000. A lower value provokes an earlier initialization. In case of signatures carrying the same value for the attribute `global_init_slot`, the sequence of their initialization rests undefined.

Example

Assume the project including 2 global variable lists `GVL_1` and `GVL_2`.

The global variable `A` is member of the exemplary global variable list `GVL_1`:

```
{attribute 'global_init_slot' := '300'}
VAR_GLOBAL
A : INT:=1000;
END_VAR
```

The initial values of the variables `B` and `C` of `GVL_2` depend on variable `A`.

```
{attribute 'global_init_slot' := '350'}
VAR_GLOBAL
B : INT:=A+1;
C : INT:=A-1;
END_VAR
```

Setting the attribute `'global_init_slot'` of `GVL_1` to 300, that is the lowest value in the exemplary initialization order, helps to ensure that the expression `A+1` is well defined when initializing `B`.

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 ([see page 596](#)) or the input assistant or the declaration part in online mode. Only the variable subsequent to the pragma will be hidden.

Syntax

```
{attribute 'hide'}
```

To hide all local variables of a signature, use the attribute `hide_all_locals` ([see page 571](#)).

Example

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 ([see page 596](#)) 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.

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 ([see page 596](#)) or the input assistant. This attribute is identical to assigning the attribute `hide` ([see page 570](#)) 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 ([see page 596](#)) 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 *SoMachine, 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

Attach the pragma `{attribute 'init_on_onlchange'}` to a variable to initialize this variable with each online change.

Syntax

```
{attribute 'init_on_onlchange' }
```

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 (*see page 593*) for the corresponding POU and the additional attribute `noinit` (*see page 586*) 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: `PLCWinNT.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.

Attribute `linkalways`

Overview

Use the pragma `{attribute 'linkalways'}` to mark POUs for the compiler in a way so that they are always included into the compile information. As a result, objects with this option will always be compiled and downloaded to the controller. This option only affects POUs and global variable lists (GVL) that are located below an application or in libraries which are inserted below an application. The compiler option **Link always** affects the same.

Syntax

```
{attribute 'linkalways'}
```

When you use the symbol configuration editor, the marked POUs are used as a basis for the selectable variables for the symbol configuration.

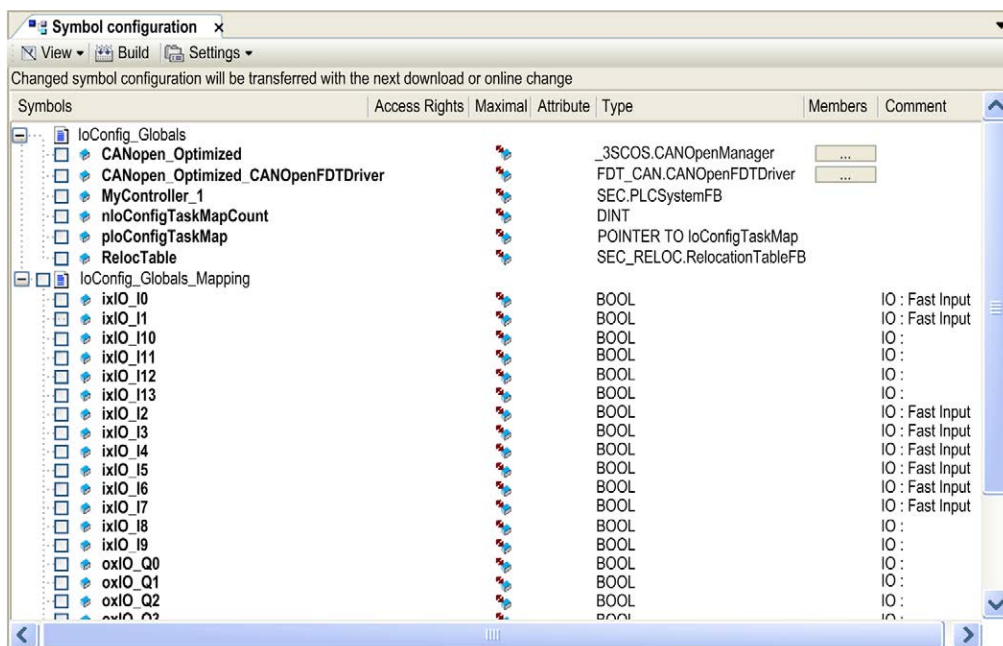
Example

The global variable list `GVLMoreSymbols` is implemented making use of the attribute `'linkalways'`:

```
{attribute 'linkalways'}  
VAR_GLOBAS  
g_iVar1: INT;  
g_iVar2: INT;  
END_VAR
```

With this code, the symbols of `GVLMoreSymbols` becomes selectable in the **Symbol configuration**.

Symbol configuration editor



Attribute monitoring

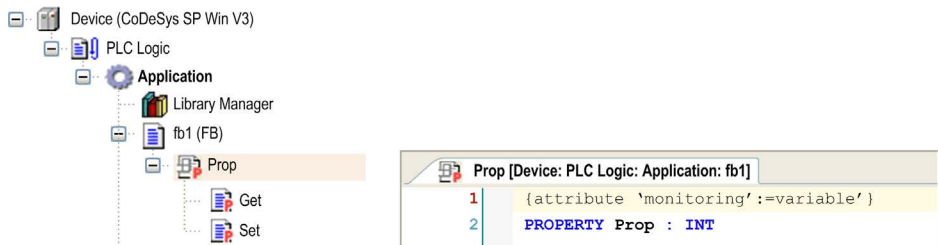
Overview

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 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

Device.Application.PLC_PRG				
Expression	Comment	Type	Value	Prepared value
fbinst		fb1		
seconds		INT	22	
milli		INT	0	
testvar		INT	22	

1	fbinst.seconds 22 := 22;
2	testvar 22 := fbinst.seconds 22; RETURN

Monitoring the Current Value of the Property Variables

There are 2 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 retrieved by a direct call of property: the monitoring service of the runtime system calls the get method and the property function will be executed.

NOTE: When choosing this monitoring type instead of using an intermediate variable (see 1. *Pragma*), consider possible side effects, which can occur if any operations on the variable are implemented within the property.

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

```
PLC_PRG
1 PROGRAM PLC_PRG
2 VAR
3     nResult2, nResult3 : INT;
4     xResult2 : BOOL;
5     xResult4: BOOL;
6 END_VAR

1 // monitoring possible:
2 nResult2 := FUN2(64,GVL.VALUE2);
3 xResult2 := FUN_BOOL2(GVL.BOOLFALSE, GVL.BOOLTRUE);
4
5 // monitoring not possible (TRUE and FALSE cannot be interpreted directly)
6 FUN_BOOL2(TRUE, FALSE);
```

Example 3: Function calls in online mode:

PLC_PRG FUN2 FUN_BOOL2

Device.Application.PLC_PRG

Expression	Type	Value	Prepared value	Comment
nResult2	INT	32		
nResult3	INT	<???		
xResult2	BOOL	FALSE		
xResult4	BOOL	<???		

1 // Monitoring possible:
2 nResult2 32 := FUN2 ??? (64,GVL.VALUE2 ???);
3 nResult3 ??? := FUN3 ??? (64,GVL.VALUE2 ??? ,GVL.VALUE3 ???);
4 xResult2 FALSE := FUN_BOOL2 ??? (GVL.BOOLFALSE ??? , GVL.BOOLTRUE ???);
5 xResult4 ??? := FUN_BOOL4 ??? (GVL.BOOLTRUE ??? , GVL.BOOLFALSE ??? , GVL.BOOL
6
7 // Monitoring not possible, True und FALSE cannot be Interpreted directly
8 FUN_BOOL2 ??? (TRUE, FALSE); RETURN

Watch 1

Expression	Type	Value	Prepared value	Comment
Device.Application.PLC_PRG.nResult2	INT	32		
Device.Application.PLC_PRG.xResult2	BOOL	FALSE		

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 namespace

Overview

In combination with the attribute symbol ([see page 594](#)), 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 SoMachine visualization also in a later SoMachine 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>
```

Attribute no_check

Overview

The pragma `{attribute no_check}` added to 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.

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

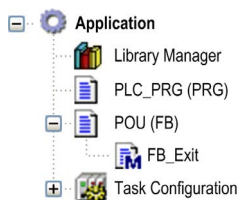
If a function block provides an exit method (*see page 549*), you can suppress its call for a special instance with the help of assigning the pragma `{attribute no-exit}` to the function block instance.

Syntax

```
{attribute 'symbol'= 'no-exit'}
```

Example

Assume the exit 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 'symbol' := 'no-exit'}
POU2 : POU;
END_VAR
  
```

When variable `bInCopyCode` becomes `TRUE` within `POU1`, the exit method `FB_Exit` is called exiting an instance that will get copied afterwards (online change), though the value of variable `bInCopyCode` will have no influence on `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_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. This means that the base class actions may be overridden by specific implementations related to the derived classes.

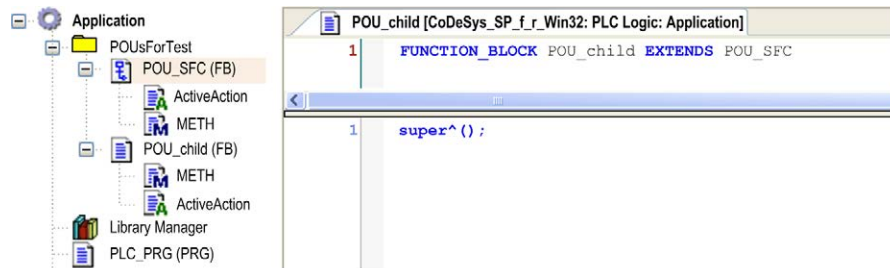
In order to help to prevent the action of the base class from being overridden, you can assign the pragma {attribute 'no_virtual_actions'} to the base class.

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 screenshot displays the Siemens STEP 7 LAD editor. The main window shows the 'POU_SFC [CoDeSys_SP_f_r_Win32]' function block. The code is as follows:

```
1 FUNCTION_BLOCK POU_SFC
2 VAR_OUTPUT
3   test_meth: STRING:='';
4   test_act: STRING:='';
5   an_int:INT:=0;
6 END_VAR
```

Below the code, a state transition diagram is visible, showing an 'Init' step leading to a 'Step0' step, with a 'true' condition for the transition. The 'Properties' window on the right shows the 'Common' tab with the following values:

Property	Value
Name	Step0
Comment	
Symbol	

The 'Specific' tab is also visible, showing the 'Initial step' checkbox is checked. The 'Times' tab shows 'Minimal a...' and 'Maximal...' fields. The 'Actions' tab shows 'Step active' as 'ActiveAction', 'Step entry' as an empty field, and 'Step exit' as an empty field.

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:

PLC_PRG [CoDeSys_SP_f_r_Win32: SPS-Logik: Applicat		
CoDeSys_SP_f_r_Win32.Application.PLC_PRG		
Expression	Type	Value
Child	POU_child	
test_meth	STRING	'child_method'
test_act	STRING	'child_action'
an_int	INT	53

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'.

PLC_PRG [CoDeSys_SP_f_r_Win32: SPS-Logik: Applicat		
CoDeSys_SP_f_r_Win32.Application.PLC_PRG		
Expression	Type	Value
Child	POU_child	
test_meth	STRING	'child_method'
test_act	STRING	'father_action'
an_int	INT	204

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 ([see page 553](#)), 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.

Syntax

```
{attribute 'pack_mode' := '<value>'}
```

Replace the template `<value>` included in single quotes by one of the following values available:

Value	Assigned Pack Mode
0	aligned (there will be no memory gaps)
1	1-byte-aligned (identical to aligned)
2	2-byte-aligned (the maximum size of a memory gap is 1 byte)
4	4-byte-aligned (the maximum size of a memory gap is 3 bytes)
8	8-byte-aligned (the maximum size of a memory gap is 7 bytes)

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` will be 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 `0x0105`.

NOTE: You can also apply the attribute to POU's. Use this application carefully due to eventual existing internal pointers of the POU.

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

The pragma {attribute 'reflection'} is attached to signatures. Due to performance reasons, it is an obligatory attribute for POUs carrying the instance-path attribute (*see page 575*).

Syntax

```
{attribute 'reflection'}
```

Example

Refer to the attribute instance-path example (*see page 575*).

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 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 exported as symbols into a symbol list.
- 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**.

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 warning disable

Overview

You can use the pragma `warning disable` to suppress alerts. To enable the display of the alert, use the pragma `warning restore`.

Syntax

```
{warning disable <compiler ID>}
```

Every alert 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 ...
C0196: Implicit conversion from unsigned Type 'UINT' to signed Type 'INT' : 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 alert will be detected for test2. But no alert will be detected for test1.

Section 27.6

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 (activated or deactivated) in the **SmartCoding** section of the **Tools → Options** dialog box.

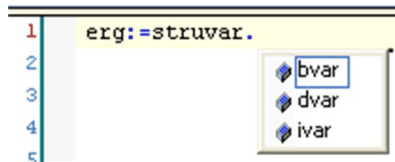
Support in Identifier Insertion

The smart coding functionality helps to insert a correct identifier:

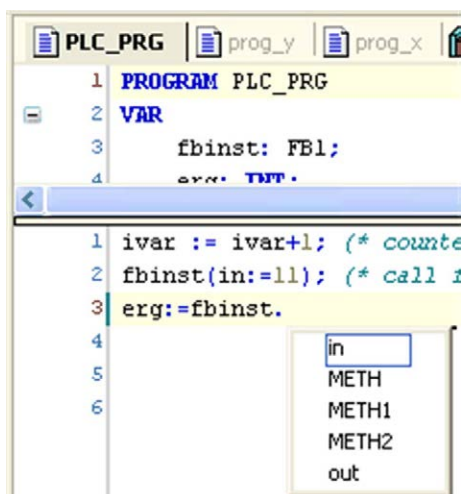
- If you - at any place, where a global identifier can be inserted - insert a dot (.) instead of the identifier, a selection box will display. It lists the currently available global variables. You can choose one of these elements and press the RETURN key to insert it behind the dot. You can also insert 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 POU's 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:



Chapter 28

Data Types

What Is in This Chapter?

This chapter contains the following sections:

Section	Topic	Page
28.1	General Information	600
28.2	Standard Data Types	601
28.3	Extensions to IEC Standard	604
28.4	User-Defined Data Types	612

Section 28.1

General Information

Data Types

Overview

You can use standard data types ([see page 601](#)), user-defined data types ([see page 612](#)), or instances of function blocks when programming in SoMachine. 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.

Section 28.2

Standard Data Types

Standard Data Types

Overview

SoMachine supports all data types (*see page 600*) described by standard IEC61131-3.

The following data types are described in this chapter:

- BOOL (*see page 601*)
- Integer (*see page 601*)
- REAL / LREAL (*see page 602*)
- STRING (*see page 603*)
- Time Data Types (*see page 603*)

Additionally, some standard-extending data types (*see page 604*) are supported and you can define your own user-defined data types (*see page 612*).

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* (*see page 733*).

NOTE: You can use implicit checks to validate the conversion of variable types (refer to the chapter *POUs for Implicit Checks* (*see page 180*)).

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

Data Type	Lower Limit	Upper Limit	Memory Space
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 ([see page 739](#)).

NOTE: You can use implicit checks to validate the conversion of variable types (refer to the chapter *POUs for Implicit Checks* ([see page 180](#))).

REAL / LREAL

The data types REAL and LREAL are so-called floating-point types. They represent rational numbers. 32 bits of memory space is reserved for REAL and 64 bits for LREAL.

Value range for REAL:

1.401e-45...3.403e+38

Value range for LREAL:

2.2250738585072014e-308...1.7976931348623158e+308

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 ([see page 685](#)) such as the following:

```
i1 := REAL_TO_INT(r1);
```

For further information, refer to REAL/LREAL constants (operands) ([see page 740](#)).

NOTE: You can use implicit checks to validate the conversion of variable types (refer to the chapter *POUs for Implicit Checks* ([see page 180](#))).

STRING

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 characters in the string and can be placed in parentheses or square brackets. If no size specification is given, the default size of 80 characters will be used.

In general, the length of a string is not limited. But string functions can only process strings with a length of 1...255 characters. 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.

NOTE: 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.

Example of a string declaration with 35 characters:

```
str:STRING(35):='This is a String';
```

For further information, refer to WSTRING ([see page 606](#)) and STRING Constants (Operands) ([see page 741](#)).

NOTE: You can use implicit checks to validate the conversion of variable types (refer to the chapter *POUs for Implicit Checks* ([see page 180](#))).

Time Data Types

The data types TIME, TIME_OF_DAY (shortened TOD), DATE, and DATE_AND_TIME (shortened DT) are handled internally like DWORD. 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.

For further information, refer to the following descriptions:

- Data Types ([see page 600](#))
- LTIME ([see page 605](#)): extension to the IEC 61131-3 standard, available as a 64-bit time data type
- TIME constants ([see page 734](#))
- DATE constants ([see page 736](#))
- DATE_AND_TIME constants ([see page 737](#))
- TIME_OF_DAY constants ([see page 738](#))

NOTE: You can use implicit checks to validate the conversion of variable types (refer to the chapter *POUs for Implicit Checks* ([see page 180](#))).

Section 28.3

Extensions to IEC Standard

Overview

This chapter lists the data types that are supported by SoMachine in addition to the standard IEC 61131-3.

What Is in This Section?

This section contains the following topics:

Topic	Page
UNION	605
LTIME	605
WSTRING	606
BIT	606
References	607
Pointers	609

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. This means that they occupy the same storage location. Thus, assuming a union definition as shown in the following example, an assignment to name . a also manipulates name . b.

Example

```
TYPE name: UNION
a : LREAL;
b : LINT;
END_UNION
END_TYPE
```

LTIME

Overview

As an extension to the IEC 61131-3, LTIME is supported as time base for high resolution timers. LTIME is of size 64 bit and resolution nanoseconds.

Syntax

LTIME#<time declaration>

The time declaration can include the time units as used with the TIME constant and as:

- us : microseconds
- ns : nanoseconds

Example

```
LTIME1 := LTIME#1000d15h23m12s34ms2us44ns
```

Compare to TIME size 32 bit and resolution milliseconds (*see page 603*).

WSTRING

Overview

This string data type is an extension to the IEC 61131-3 standard.

It differs from the standard STRING type (ASCII) by interpretation in Unicode format, and needing 2 bytes for each character and 2 bytes extra memory space (each only 1 in case of a STRING).

Example

```
wstr:WSTRING:="This is a WString";
```

For further information, refer to the following descriptions:

- STRING ([see page 603](#))
- STRING constants ([see page 741](#)) (operands)

BIT

Overview

You can use the BIT data type only for particular variables within Structures ([see page 617](#)). 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* ([see page 618](#))). Bit elements which are declared one after another will be combined in bytes. In contrast to BOOL types ([see page 601](#)), 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 is an alias for an object. The alias can be written or read via identifiers. The difference to a pointer is that the value pointed to is directly affected and that the assignment of reference and value is fixed. Set the address of the reference via a separate assignment operation. You can use the operator `__ISVALIDREF` to verify whether a reference points to a valid value (that is unequal to 0). For further information, refer to the paragraph *Check for Valid References* further below in this chapter.

Syntax

<identifier> : REFERENCE TO <data type>

Example Declaration

```
ref_int : REFERENCE TO INT;
a : INT;
b : INT;
```

ref_int is now available for being used as an alias for variables of type INT.

Example of Use

```
ref_int REF= a;    (* ref_int now points to a *)
ref_int := 12;     (* a now has value 12 *)
b := ref_int * 2;  (* b now has value 24 *)
ref_int REF= b;    (* ref_int now points to b *)
ref_int := a / 2;  (* b now has value 6 *)
ref_int REF= 0;    (* explicit initialization of the reference *)
```

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> := __ISVALIDREF(identifier, declared with type <REFERENCE TO <datatype>);
<boolean variable> will be TRUE, if the reference points to a valid value, FALSE if not.
```

Example

Declaration

```
ivar : INT;  
ref_int : REFERENCE TO INT;  
ref_int0 : REFERENCE TO INT;  
testref: BOOL := FALSE;
```

Implementation

```
ivar := ivar +1;  
ref_int REF= hugo;  
ref_int0 REF= 0;  
testref := __ISVALIDREF(ref_int); (* will be TRUE, because ref_int po  
ints to ivar, which is unequal 0 *)  
testref0 := __ISVALIDREF(ref_int0); (* will be FALSE, because ref_int0  
is set to 0 *)
```

Pointers

Overview

As an extension to the IEC 61131-3 standard, you can use pointers.

Pointers save the addresses of variables, programs, function blocks, methods, and functions while an application program is running. A pointer can point to any of those objects and to any data type (*see page 600*), even to user-defined data types (*see page 613*). The possibility of using an implicit pointer check function is described further below in the paragraph *CheckPointer function* (*see page 610*).

Syntax of a Pointer Declaration

<identifier>: POINTER TO <data type | function block | program | method | function>;

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) (*see page 682*) after the pointer identifier. See pt^ in the example below.

You can use the ADR address operator (*see page 681*) to assign the address of a variable to a pointer.

Example

```
VAR
  pt:POINTER TO INT; (* of pointer pt *)
var_int1:INT := 5; (* declaration of variables var_int1 and var_int2 *)
  var_int2:INT;
END_VAR
pt := ADR(var_int1); (* address of var_int1 is assigned to pointer pt *)
var_int2:= pt^;      (* value 5 of var_int1 gets assigned to var_int2 via dereferencing of pointer pt; *)
```

Function Pointers

SoMachine also supports function pointers. 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 (*see page 681*) 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` (*see page 603*) and `WSTRING` (*see page 606*) is allowed.

- `ptr[i]` returns the base data type.
- Index access on pointers is arithmetic:
If the index access is used on a variable of type pointer, the offset `ptr[i]` equates to $(ptr + i * \text{sizeof}(\text{base type}))^\wedge$. The index access also performs an implicit dereferencing on the pointer. The result type is the base type of the pointer.
Consider that `ptr[7]` does not equate to $(ptr + 7)^\wedge$.
- If the index access is used on a variable of type `STRING`, the result is the character at offset `index - expr`. The result is of type `BYTE`. `str[i]` will return the *i*-th character of the string as a `SINT` (ASCII).
- If the index access is used on a variable of type `WSTRING`, the result is the character at offset `index - expr`. The result is of type `WORD`. `wstr[i]` will return the *i*-th character of the string as `INT` (Unicode).

NOTE: You can also use References (*see page 607*). In contrast to a pointer, references directly affect a value.

CheckPointer Function

For checking pointer access during runtime, you can use the implicitly available check function `CheckPointer`. It is called before each access on the address of a pointer. To achieve this, add the object **POUs for implicit checks** to the application. Do this by activating the checkbox related to the type **CheckPointer**, choosing an implementation language, and confirming your settings by clicking **Open**. This opens the check function in the editor corresponding to the implementation language selected. Independently of this choice, the declaration part is preset. You cannot modify it except of adding further local variables. However, in contrast to other check functions, there is no default implementation of `CheckPointer` available.

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
```

Implementation part: (incomplete):

```
// No standard way of implementation. Fill your own code here
CheckPointer := ptToTest;
```

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.

In case of a positive result of the check, the unmodified input pointer will be returned (ptToTest).

Section 28.4

User-Defined Data Types

What Is in This Section?

This section contains the following topics:

Topic	Page
Defined Data Types	613
Arrays	614
Structures	617
Enumerations	619
Subrange Types	621

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 in the **POUs tree** or **Devices tree** or within the declaration part of a POU.

See the recommendations on the naming of objects ([see page 522](#)) in order to make it as unique as possible.

See the following user-defined data types:

- arrays ([see page 614](#))
- structures ([see page 617](#))
- enumerations ([see page 619](#))
- subrange types ([see page 621](#))
- references ([see page 607](#))
- pointers ([see page 609](#))

Arrays

Overview

One, two and three-dimensional fields (arrays) are supported as elementary data types. 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 ([see page 615](#)).

Syntax

<Array_Name>:ARRAY [<ll1>..

ll1, ll2, ll3 identify the lower limit of the field range.

ul1, ul2 and ul3 identify the upper limit of the field range.

The range values have to be of type integer.

Example

```
Card_game: ARRAY [1..13, 1..4] OF INT;
```

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[6] . . . arr1[10]` are therefore initialized with 0.

Accessing Array Components

In a two-dimensional array, access the components 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. Therefore, add the object **POUs for implicit checks** to the application using **Add Object → POU's for implicit checks**. Select the checkbox related to the type **CheckBounds**. Choose an implementation language. Confirm your settings with **Open**. The function `CheckBound` will be opened in the editor corresponding to the implementation language selected. Independently of this choice, the declaration part is preset. You cannot modify it except for adding further local variables. The ST editor gives a proposal default implementation of the function that you can modify.

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 will be called implicitly as soon as a variable of type `ARRAY` 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 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 case, 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 will be assigned to the element a[7] of the array. This is how you can correct attempted access outside the field range via the function CheckBounds.

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 Polygonline:
STRUCT
    Start:ARRAY [1..2] OF INT;
    Point1:ARRAY [1..2] OF INT;
    Point2:ARRAY [1..2] OF INT;
    Point3:ARRAY [1..2] OF INT;
    Point4:ARRAY [1..2] OF INT;
    End:ARRAY [1..2] OF INT;
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* ([see page 614](#)).

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` (*see page 606*) 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 string constants. These constants are referred to as enumeration values.

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

TYPE <identifier> (<enum_0>,<enum_1>, ...,<enum_n>) |<base data type>;END_TYPE

A variable of type <identifier> can take on one of the enumeration values <enum_. .> and will be initialized with the first one. These values are compatible with whole numbers which means that you can perform operations with them just as you would do with integer variables. You can assign a number x to the variable. If the enumeration values are not initialized with specific values within the declaration, counting will begin with 0. When initializing, ensure that the initial values are increasing within the row of components. The validity of the number is verified at the time it is run.

Example

```
TYPE TRAFFIC_SIGNAL: (red, yellow, green:=10); (* The initial value for
each of the colors is red 0, yellow 1, green 10 *)
END_TYPE
TRAFFIC_SIGNAL1 : TRAFFIC_SIGNAL;
TRAFFIC_SIGNAL1:=0; (* The value of the traffic signal is "red" *)
FOR i:= red TO green DO
  i := i + 1;
END_FOR;
```

First Extension to the IEC 61131-3 Standard

You can use the type name of enumerations (as a scope operator ([see page 728](#))) 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

```
TYPE COLORS_1: (red, blue);
END_TYPE
TYPE COLORS_2: (green, blue, yellow);
END_TYPE
```

Use of enumeration value blue in a POU

Declaration

```
colorvar1 : COLORS_1;  
colorvar2 : COLORS_2;
```

Implementation

```
(* possible: *)  
colorvar1 := colors_1.blue;  
colorvar2 := colors_2.blue;  
(* not possible: *)  
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

The base data type for enumeration BigEnum should be DINT:

```
TYPE BigEnum : (yellow, blue, green:=16#8000) DINT;  
END_TYPE
```


Subrange Types

Overview

A subrange type is a user-defined type (*see page 613*) whose range of values is only a subset of that of the basic data type. You can also use implicit range boundary checks (*see page 622*).

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 basic type, setting the lower boundary of the range types The lower boundary itself is included in this range.
<og>	a constant compatible with the basic type, setting the upper boundary of the range types. The upper boundary itself is included in this basic 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 check the range limits during runtime, the functions `CheckRangeSigned` or `CheckRangeUnsigned` have to be available to the application. You can add the object **POUs for implicit checks** to the application using the **Add Object** dialog box. Mark the checkbox related to the type **CheckRangeSigned** or **CheckRangeUnsigned**. Choose an implementation language. Confirm your settings with **Open**. The selected function will be opened in the editor corresponding to the implementation language selected. Independently of this choice, the declaration part is preset. You cannot modify it except for adding further local variables. The ST editor proposes a default implementation of the function that you can modify.

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 `CheckRangeSigned` or `CheckRangeUnsigned` is present, no type checking of subrange types occurs during runtime. In this case, variable `i` could get any value between `-32768` and `32767` at any time.

Chapter 29

Programming Guidelines

What Is in This Chapter?

This chapter contains the following sections:

Section	Topic	Page
29.1	Naming Conventions	626
29.2	Prefixes	628

Section 29.1

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

Recommended Designator	Not Recommended Designators
diState	diSTATE
xInit	x_Init
diCycleCounter	diCyclecounter
lrRefVelocity	lrRef_Velocity
c_lrMaxPosition	c_lrMaxPosition
FC_PidController	FC_PIDController

Section 29.2

Prefixes

What Is in This Section?

This section contains the following topics:

Topic	Page
Prefix Parts	629
Order of Prefixes	630
Scope Prefix	632
Data Type Prefix	633
Property Prefix	635
POU Prefix	636
Namespace Prefix	637

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 (<i>see page 632</i>)	scope of variables and constants	[scope]_[designator]	G_diFirstUserFault
data type prefix (<i>see page 633</i>)	identifying the data type of variables and constants	[type][designator]	xEnable
property prefix (<i>see page 635</i>)	identifying the properties of variables and constants	[property]_[designator]	c_iNumberOfAxes
POU prefix (<i>see page 636</i>)	identifying if POU was implemented as a function, function block, or program	[POU]_[designator]	FB_VisuController
namespace prefix (<i>see page 637</i>)	for POU, 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
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

Data type prefix	Type	Use (memory location)	Example
st	struct	structure	stName
if	interface	interface	ifMotion
ut	union	union	uName
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

The designator is composed of a POU prefix and as short a name as possible (for example, FB_GetResult). Just like a variable, capitalize the first letter of each word in the basis 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

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();
```

Chapter 30

Operators

Overview

SoMachine supports all IEC operators. 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 prescribed by the standard:

- ANDN
- ORN
- XORN
- SIZEOF (refer to arithmetic operators (*see page 640*))
- ADR
- BITADR
- content operator (refer to address operators (*see page 680*))
- some scope operators (*see page 727*)

What Is in This Chapter?

This chapter contains the following sections:

Section	Topic	Page
30.1	Arithmetic Operators	640
30.2	Bitstring Operators	654
30.3	Bit-Shift Operators	659
30.4	Selection Operators	667
30.5	Comparison Operators	673
30.6	Address Operators	680
30.7	Calling Operator	684
30.8	Type Conversion Operators	685
30.9	Numeric Functions	702
30.10	IEC Extending Operators	715
30.11	Initialization Operator	729

Section 30.1

Arithmetic Operators

Overview

The following operators, prescribed 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.

What Is in This Section?

This section contains the following topics:

Topic	Page
ADD	641
MUL	643
SUB	645
DIV	647
MOD	650
MOVE	652
SIZEOF	653

ADD

Overview

IEC operator for the addition of variables

Allowed types

- BYTE
- WORD
- DWORD
- LWORD
- SINT
- USINT
- INT
- UINT
- DINT
- UDINT
- LINT
- ULINT
- REAL
- LREAL
- TIME
- TIME_OF_DAY(TOD)
- DATE
- DATE_AND_TIME(DT)

For time data types, the following combinations are possible:

- TIME+TIME=TIME
- TOD+TIME=TOD
- DT+TIME=DT

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.

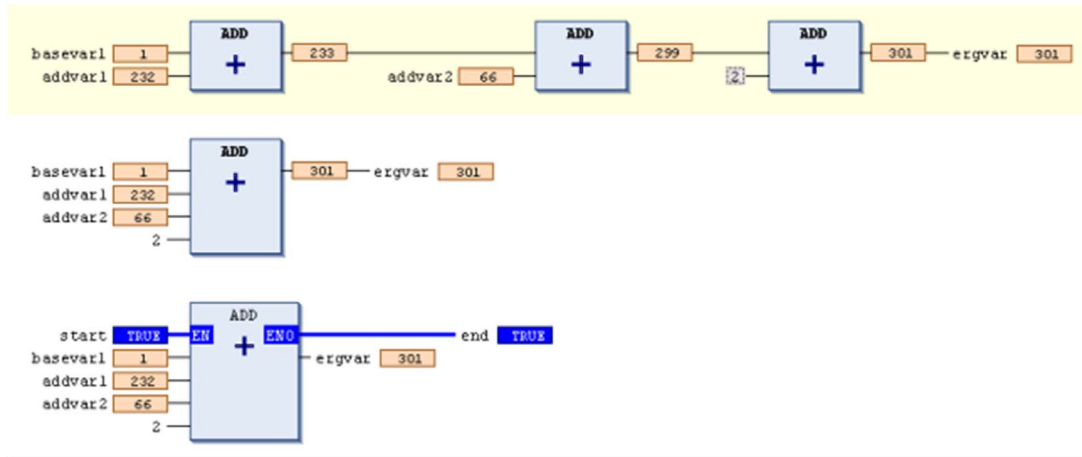
Example in IL

```
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.

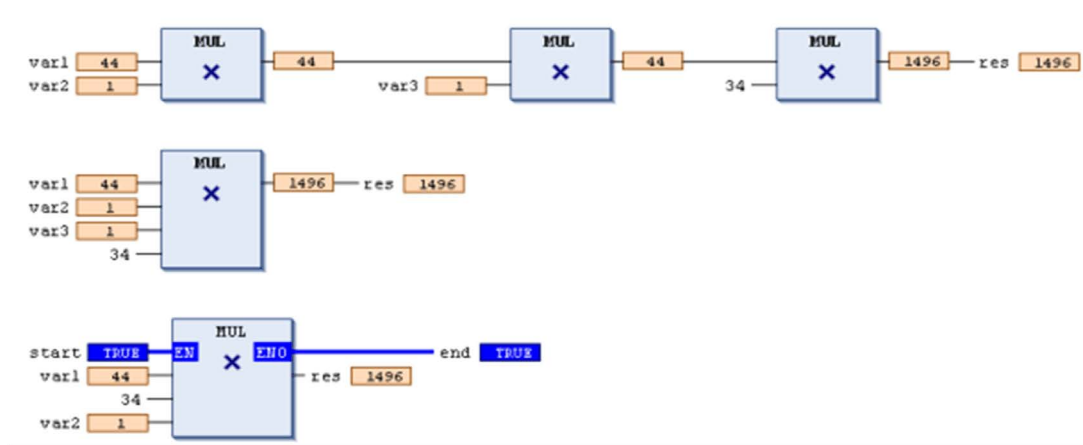
Example in IL

```
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
- TIME_OF_DAY(TOD)
- DATE
- DATE_AND_TIME(DT)

For time data types, the following combinations are possible:

- TIME-TIME=TIME
- DATE-DATE=TIME
- TOD-TIME=TOD
- TOD-TOD=TIME
- DT-TIME=DT
- DT-DT=TIME

Consider that negative TIME 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

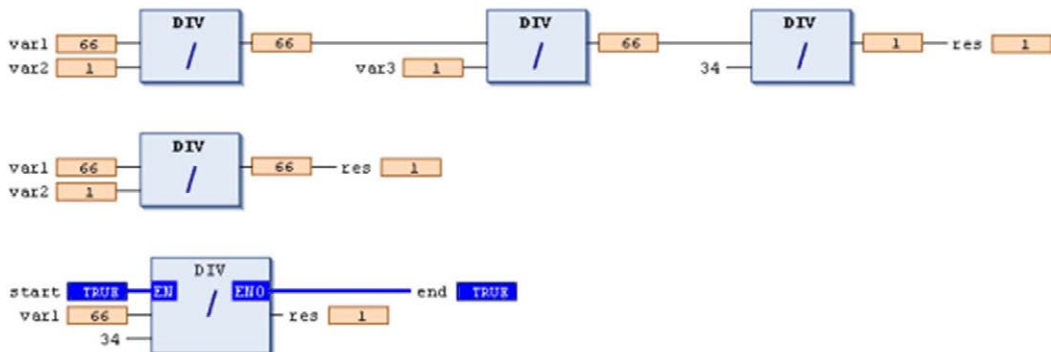
(Result in Var1 is 4.)

```
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* ([see page 180](#)).

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:

- CheckDivInt
- CheckDivLint
- CheckDivReal
- CheckDivLReal

For information on inserting the function, refer to the description of the **POUs for implicit checks** function ([see page 180](#)).

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* ([see page 180](#)).

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
$0 \leq \text{size of } x < 256$	USINT
$256 \leq \text{size of } x < 65,536$	UINT
$65,536 \leq \text{size of } x < 4,294,967,296$	UDINT
$4,294,967,296 \leq \text{size of } x$	ULINT

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
```

Section 30.2

Bitstring Operators

Overview

The following bitstring operators are available, matching the IEC1131-3 standard:

- AND (*see page 655*)
- OR (*see page 656*)
- XOR (*see page 657*)
- NOT (*see page 658*)

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.

What Is in This Section?

This section contains the following topics:

Topic	Page
AND	655
OR	656
XOR	657
NOT	658

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.

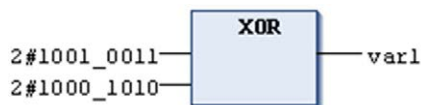
```
Var1:BYTE;
```

```
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



Section 30.3

Bit-Shift Operators

What Is in This Section?

This section contains the following topics:

Topic	Page
SHL	660
SHR	662
ROL	663
ROR	665

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 is noticed 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 noticed. 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
```

Section 30.4

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 *(see page 668)*
- MAX *(see page 669)*
- MIN *(see page 670)*
- LIMIT *(see page 671)*
- MUX *(see page 672)*

What Is in This Section?

This section contains the following topics:

Topic	Page
SEL	668
MAX	669
MIN	670
LIMIT	671
MUX	672

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, OUT) : any type

G: BOOL

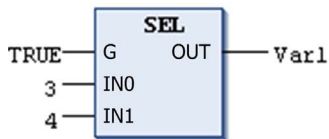
Example in IL

```
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: An expression occurring ahead of IN1 will not be processed if IN0 is TRUE. An expression occurring ahead of IN2 will not be processed if IN0 is FALSE.

MAX

Overview

IEC selection operator performing a maximum function.

The MAX operator returns the greater of the 2 values.

```
OUT := MAX(IN0, IN1)
```

IN0, IN1 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

```
Var1:=MAX(30,40); (* Result is 40 *)
Var1:=MAX(40,MAX(90,30)); (* Result is 90 *)
```

Example in FBD



MIN

Overview

IEC selection operator performing a minimum function.

The MIN operator returns the lesser of the 2 values.

```
OUT := MIN(IN0, IN1)
```

IN0, IN1 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); (* Result is 30 *);
Var1:=MIN(MIN(90,30),40); (* Result is 30 *);
```

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 type of variable.

`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: An expression occurring ahead of an input other than `INk` will not be processed to save run time. Only in simulation mode will all expressions be executed.

Section 30.5

Comparison Operators

Overview

The following operators matching the IEC1131-3 standard are available:

- GT (*see page 674*)
- LT (*see page 675*)
- LE (*see page 676*)
- GE (*see page 677*)
- EQ (*see page 678*)
- NE (*see page 679*)

What Is in This Section?

This section contains the following topics:

Topic	Page
GT	674
LT	675
LE	676
GE	677
EQ	678
NE	679

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.

The operands can be of any basic data type.

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.

The operands can be of any basic data type.

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.

The operands can be of any basic data type.

Example in IL

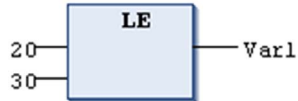
Result is TRUE

```
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.

The operands can be of any basic data type.

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.

The operands can be of any basic data type.

Example in IL

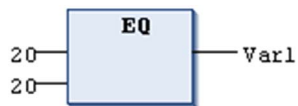
Result is TRUE

```
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.

The operands can be of any basic data type.

Example in IL

```
LD    40
NE    40
ST    Var1
```

Example in ST

```
VAR1 := 40 <> 40;
```

Example in FBD



Section 30.6

Address Operators

What Is in This Section?

This section contains the following topics:

Topic	Page
ADR	681
Content Operator	682
BITADR	683

ADR

Overview

This address operator is not specified by the standard IEC 61131-3.

ADR returns the address (*see page 747*) of its argument in a DWORD. This address can be assigned to a pointer (*see page 609*) within the project.

NOTE: SoMachine allows you to use the ADR operator with function names, program names, function block names, and method names.

Refer to the chapter *Pointers* (*see page 609*) and consider that function pointers can be passed to external libraries. Nevertheless, there is no possibility to call a function pointer within SoMachine. 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

```
dwVar :=ADR( bVar );
```

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 variables point to invalid memory. So, ensure that a pointer is not kept between cycles, but is reassigned in each cycle.

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: 16x40000000

Input: 16x80000000

Output: 16xC0000000

Example in ST

```
VAR
    var1 AT %IX2.3:BOOL;
    bitoffset: DWORD;
END_VAR
bitoffset:=BITADR(var1); (* Result if byte addressing=TRUE: 16x80000013
, if byte addressing=FALSE: 16x80000023 *)
```

Example in IL

```
LD    Var1
BITADR
ST    bitoffset
```

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.

Section 30.7

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)
```


Section 30.8

Type Conversion Operators

What Is in This Section?

This section contains the following topics:

Topic	Page
Type Conversion Functions	686
BOOL_TO Conversions	687
TO_BOOL Conversions	689
Conversion Between Integral Number Types	691
REAL_TO / LREAL_TO Conversions	692
TIME_TO/TIME_OF_DAY Conversions	694
DATE_TO/DT_TO Conversions	696
STRING_TO Conversions	698
TRUNC	700
TRUNC_INT	701
ANY_ . . . _TO Conversions	701

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

<elem.type1>_TO_<elem.type2>

NOTE: At ...TO_STRING conversions the string is generated as left-justified. If it is defined too short, it will be cut from the right side.

The following type conversions are supported:

- BOOL_TO conversions ([see page 687](#))
- TO_BOOL conversions ([see page 689](#))
- conversion between integral number types ([see page 691](#))
- REAL_TO-/ LREAL_TO conversions ([see page 692](#))
- TIME_TO/TIME_OF_DAY conversions ([see page 694](#))
- DATE_TO/DT_TO conversions ([see page 696](#))
- STRING_TO conversions ([see page 698](#))
- TRUNC ([see page 700](#)) (conversion to DINT)
- TRUNC_INT ([see page 701](#))
- ANY_NUM_TO_<numeric data type>
- ANY_..._TO conversions ([see page 701](#))

BOOL_TO Conversions

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

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

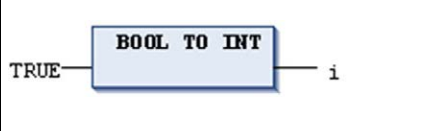
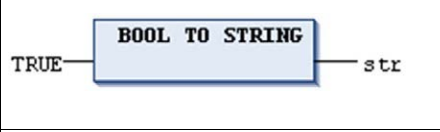
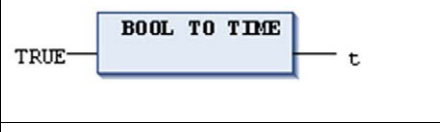
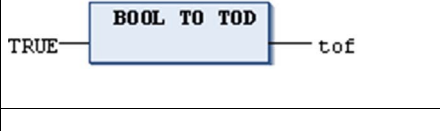
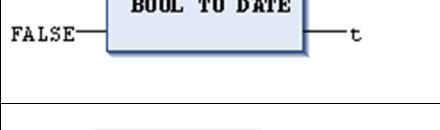
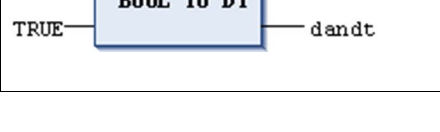
Examples in IL with conversion results:

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

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

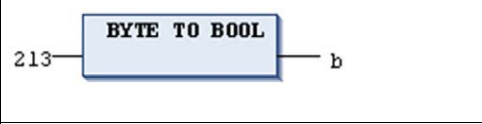
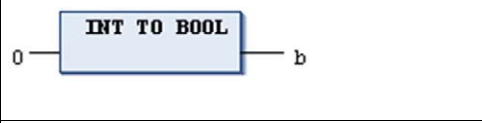
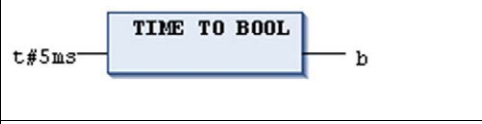
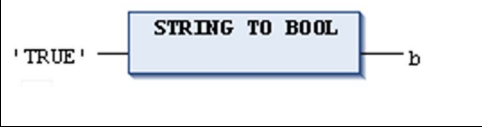
Examples in IL

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
 213 — BYTE TO BOOL — b	TRUE
 0 — INT TO BOOL — b	FALSE
 t#5ms — TIME TO BOOL — b	TRUE
 'TRUE' — STRING TO BOOL — b	TRUE

Conversion Between Integral Number Types

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* ([see page 601](#)).

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

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

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 16. If the (L)REAL number has more digits, then the sixteenth 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

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.

Examples in ST

Examples in ST with conversion results:

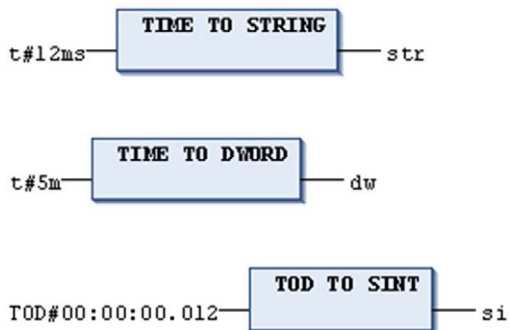
Example	Result
str := TIME_TO_STRING(T#12ms);	T#12ms
dw := TIME_TO_DWORD(T#5m);	300000
si := TOD_TO_SINT(TOD#00:00:00.012);	12

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

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.

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

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) (*see page 541*) of the target type. This applies to the specification of exponential values, infinite values, prefixes, grouping character ("_") and comma. Additional characters after the digits of a number are allowed, as for example, 23xy. Characters preceding a number are not allowed.

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
<code>b := STRING_TO_BOOL('TRUE');</code>	TRUE
<code>w := STRING_TO_WORD('abc34');</code>	0
<code>w := STRING_TO_WORD('34abc');</code>	34
<code>t := STRING_TO_TIME('T#127ms');</code>	T#127ms
<code>r := STRING_TO_REAL('1.234');</code>	1.234
<code>bv := STRING_TO_BYTE('500');</code>	244

Example in FBD

Example	Conversion result
 <pre> graph LR A["'TRUE'"] --> B["STRING TO BOOL"] B --> C["b"] </pre>	TRUE

TRUNC

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 of 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

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 a DINT or INT. The behavior of 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

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)

Section 30.9

Numeric Functions

Overview

This chapter describes the available numeric IEC operators specified by the standard IEC 61131-3.

What Is in This Section?

This section contains the following topics:

Topic	Page
ABS	703
SQRT	704
LN	705
LOG	706
EXP	707
SIN	708
COS	709
TAN	710
ASIN	711
ACOS	712
ATAN	713
EXPT	714

ABS

Definition

Numeric IEC operator for returning the absolute value of a number.

In- and output can be of any numeric basic data type.

Example in IL

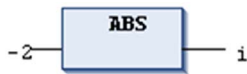
The result in `i` is 2.

```
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 basic 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 basic 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 basic 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 basic 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 can be of any numeric basic data type, the output variable has to be of type REAL or LREAL.

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 can be of any numeric basic data type where the output variable has to be of type REAL or LREAL.

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 basic 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 basic 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 basic 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 basic 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 any numeric basic data type 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



Section 30.10

IEC Extending Operators

What Is in This Section?

This section contains the following topics:

Topic	Page
IEC Extending Operators	716
__DELETE	717
__ISVALIDREF	719
__NEW	720
__QUERYINTERFACE	723
__QUERYPOINTER	725
Scope Operators	727

IEC Extending Operators

Overview

In addition to the IEC operators, SoMachine supports following IEC extending operators:

- ADR (*see page 681*)
- BITADR (*see page 683*)
- SIZEOF (*see page 653*)
- __DELETE (*see page 717*)
- __ISVALIDREF (*see page 719*)
- __NEW (*see page 720*)
- __QUERYINTERFACE (*see page 723*)
- __QUERYPOINTER (*see page 725*)
- scope operators (*see page 727*)

__DELETE

Definition

This operator is not specified by the IEC 61131-3 standard.

NOTE: For compatibility reasons, the SoMachine compiler version must be greater than or equal to 3.1.10.1.

For further information, refer to the SoMachine/CoDeSys compiler version mapping table in the Compatibility and Migration User Guide (*see SoMachine Compatibility and Migration, User Guide*).

The __DELETE operator deallocates the memory for objects allocated before via the __NEW operator (*see page 720*).

__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 (**View → Properties...** → **Application build options**).

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.

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 (*see page 607*) data type.

__NEW

Definition

This operator is not prescribed by the IEC 61131-3 standard.

NOTE: For compatibility reasons, the SoMachine compiler version must be greater than or equal to 3.1.10.1.

For further information, refer to the SoMachine/CoDeSys compiler version mapping table in the Compatibility and Migration User Guide (*see SoMachine Compatibility and Migration, User Guide*).

The operator __NEW allocates memory for function block instances 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 will occur.

Activate the option **Use dynamic memory allocation** in the **Application build options** view (**View → Properties... → Application build options**) to use the __NEW operator.

If no memory could be allocated, by __NEW will return 0.

For deallocating, use __DELETE.

Syntax

__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 <length> 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 copy code in online change of dynamically created objects is not possible.

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:

```

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
  test1 : INT := 1234;
  _inc : INT := 0;
  _dut : POINTER TO DUT;
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);
    pFB^.INC();
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 prescribed 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.

NOTE: For compatibility reasons, the definition of the reference to be converted has to be an extension of the base interface __SYSTEM.IQueryInterface and the compiler version must be $\geq 3.3.0.20$.

For further information, refer to the SoMachine/CoDeSys compiler version mapping table in the Compatibility and Migration User Guide (*see SoMachine Compatibility and Migration, User Guide*).

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
PROGRAMM 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 = FALSE
, 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.

NOTE: For compatibility reasons, the definition of the intended interface reference has to be an extension of the base interface __SYSTEM.IQueryInterface and the compiler version must be $\geq 3.3.0.20$.

For further information, refer to the SoMachine/CoDeSys compiler version mapping table in the Compatibility and Migration User Guide (*see SoMachine Compatibility and Migration, User Guide*).

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;
```

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 to define the respective namespace:

- global scope operator
- global variable list name
- enumeration name
- library namespace

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 use the library namespace to access the library components explicitly.

Example

If a library which is included in a project contains a module `fun1` and there is also a POU `fun1` defined locally in the project, then you can add the namespace of the library to the module name in order to make the access unique.

Syntax

<namespace>.<module name>, for example lib1.fun1.

By default, the namespace of a library is identical to the library name. However, you can define another one 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**.

Example

There is a function fun1 in library lib. There is also a function fun1 declared in the project. By default, the namespace of library lib is named 'lib':

```
res1 := fun(in := 12); // call of the project function fun
res2 := lib.fun(in := 12); // call of the library function fun
```

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
```

Section 30.11

Initialization Operator

INI Operator

Overview

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` (*see page 546*). However, the operator can still be used for keeping compatibility with projects imported from earlier SoMachine versions.

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.

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.

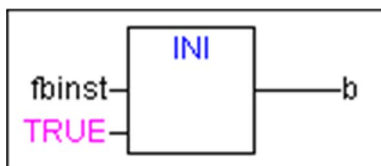
Declaration in POU

```
fbinst:fb;  
b:bool;
```

Implementation part

```
b := INI(fbinst, TRUE);  
ivar:=fbinst.retvar (* => retvar gets initialized *)
```

Example of Operator Call in FBD



Chapter 31

Operands

What Is in This Chapter?

This chapter contains the following sections:

Section	Topic	Page
31.1	Constants	732
31.2	Variables	743
31.3	Addresses	747
31.4	Functions	750

Section 31.1

Constants

What Is in This Section?

This section contains the following topics:

Topic	Page
BOOL Constants	733
TIME Constants	734
DATE Constants	736
DATE_AND_TIME Constants	737
TIME_OF_DAY Constants	738
Number Constants	739
REAL/LREAL Constants	740
STRING Constants	741
Typed Constants / Typed Literals	742

BOOL Constants

Overview

BOOL constants are the logical values TRUE and FALSE.

Refer to the description of the data type BOOL (*see page 601*).

TIME Constants

Overview

TIME constants are used to operate the standard timer modules. The time constant TIME is of size 32 bit and matches 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 resolution nanoseconds.

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: days
- h: hours
- m: minutes
- s: seconds
- 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 TIME data types ([see page 603](#)).

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 <year-month-day>.

DATE values are internally handled as DWORD values, containing the time span in seconds since 01.01.1970, 00:00 clock.

Examples

```
DATE#1996-05-06  
d#1972-03-29
```

For further information, refer to the description of the TIME data types ([see page 603](#)).

DATE_AND_TIME Constants

Overview

DATE constants and TIME_OF_DAY 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.

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 TIME data types ([see page 603](#)).

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 TIME data types ([see page 603](#)).

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. This means that a DINT variable cannot simply be used as an INT variable. Use the type conversion functions (*see page 686*).

REAL/LREAL Constants

Overview

REAL and LREAL constants can be given as decimal fractions and represented exponentially. Use the standard American format with the decimal point to do this.

NOTE: When you create a function block with an implementation of a function that includes an LREAL constant (such as IF_TouchProbe), then insert the attribute monitoring (*see page 578*).

Example:

```
{attribute 'monitoring' := 'call'}  
PROPERTY CaptureValue : LREAL
```

Examples

7.4	instead of 7,4
1.64e+009	instead of 1,64e+009

STRING Constants

Overview

A string is an arbitrary sequence of characters. STRING (*see page 603*) constants are preceded and followed by single quotation marks. You may also enter blank spaces and special characters (special characters for different languages, like accents or umlauts). They will be treated just like all other characters.

In strings, the combination of the dollar sign (\$) followed by 2 hexadecimal numbers will be interpreted as a hexadecimal representation of the 8-bit character code.

Further on, there are some combinations of characters starting with a dollar sign which are interpreted as follows:

Entered Combination	Interpretation
\$<two hex numbers>	hexadecimal representation of the 8-bit character code
\$\$	dollar sign
\$'	single quotation mark
\$L or \$l	line feed
\$N or \$n	new line
\$P or \$p	page feed
\$R or \$r	line break
\$T or \$t	tab

Examples

```
'w1Wüß?'
' Abby and Craig '
':-)'
'costs ($$)'
```

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.

Section 31.2

Variables

What Is in This Section?

This section contains the following topics:

Topic	Page
Variables	744
Addressing Bits in Variables	745

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* ([see page 517](#)) for information on the declaration of a variable, including the rules concerning the variable identifier and multiple use.

It depends on the data type ([see page 599](#)) 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 (see page 614) components
<structure name>.<variable name>	structure (see page 617) variables
<function block 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.

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, this means that 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 on BIT Data Types

The BIT data type is a special data type which is only allowed in structures. For further information, refer to Bit Access in Structures ([see page 618](#)).

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.OperationEnabled := TRUE;
```

Section 31.3

Addresses

Address

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.

Overview

When specifying an address, the memory location and size are indicated by special character sequences.

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
None	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
%IW2.5.7.1	interpretation depends on the current controller configuration (see below)
ivar AT %IW0: WORD;	example of a variable declaration including an address assignment For further information, refer to the AT Declaration chapter (<i>see page 530</i>).

Assigning Valid Addresses

For assigning a valid address within an application, specify the following:

- the appropriate position within the process image that is the memory area to be used: I = input, Q = output or M = memory area as indicated in the table above
- the desired size: X = bit, B = byte, W = word, D = dword as indicated in the example table

The current device configuration and settings (hardware structure, device description, I/O settings) play a decisive role. Especially, consider the differences in bit address interpretation between devices using byte addressing mode or those using word-oriented IEC addressing mode. For example, in a byte addressing device the first element of bit address %IX5.5 will address byte 5, but in a word addressing device it will address word 5. In contrast to that, the addressing of a word or byte address is independent of the device type: with %IW5, always word 5 will be addressed, and with byte address %IB5, always byte 5.

Depending on the size and addressing mode, different memory cells can be addressed by the same address definition.

Differences Between Byte Addressing and Word Oriented IEC Addressing

See the table below for a comparison of byte addressing and word-oriented IEC addressing for bits, bytes, words, and dwords. It visualizes the overlapping memory areas in case of byte addressing mode (see the example below the table).

Concerning the notation, consider that, for bit addresses, the IEC addressing mode is always word-oriented. This means that the place before the dot corresponds to the number of the word, the place behind names the number of the bit.

Comparison of byte and word oriented addressing for the address sizes D, W, B and X:

DWords / Words				Bytes	X (Bits)					
byte addressing		word-oriented IEC addressing			byte addressing			word-oriented IEC addressing		
D0	W0	D0	W0	B0	X0.7	...	X0.0	X0.7	...	X0.0
D1	W1	–	–	B1	X1.7	...	X1.0	X0.15	...	X0.8
...	W2	–	W1	B2	...	–	–	X1.7	...	X1.0
–	W3	–	–	B3	–	–	–	X1.15	...	X1.8
–	W4	D1	W2	B4	–	–	–	–	–	–
–	...	–	–	B5	–	–	–	–	–	–
–	–	–	W3	B6	–	–	–	–	–	–
–	–	–	–	B7	–	–	–	–	–	–
–	–	D2	W4	B8	–	–	–	–	–	–
–	–	...	–	...	–	–	–	–	–	–
–	–	...	–	...	–	–	–	–	–	–
–	–	...	–	...	–	–	–	–	–	–
D(n-3)	–	D(n/4)	...	–	–	–	–	–	–	–
–	–	–	–	–	–	–	–	–	–	–
–	W(n-1)	–	W(n/2)	–	–	–	–	–	–	–
–	–	–	–	Bn	Xn.7	...	Xn.0	X(n/2).15	...	X(n/2).8
n = byte number										

Example for overlapping of memory ranges in case of byte addressing mode:

- D0 contains B0 . . . B3
- W0 contains B0 and B1
- W1 contains B1 and B2
- W2 contains B2 and B3

In order to get around the overlap do not use W1 or D1, D2, D3 for addressing.

NOTE: Boolean values will be allocated bitwise if no explicit single-bit address is specified.

Example: A change in the value of var boo11 AT %QW0 affects the range from QX0.0 . . . QX0.7.

Section 31.4

Functions

Functions

Overview

In ST, a function call can be used as an operand.

Example

```
Result := Fct(7) + 3;
```

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();
```

Part VIII

SoMachine Templates

What Is in This Part?

This part contains the following chapters:

Chapter	Chapter Name	Page
32	General Information about SoMachine Templates	753
33	Managing Device Templates	765
34	Managing Function Templates	779

Chapter 32

General Information about SoMachine Templates

Section 32.1

SoMachine Templates

What Is in This Section?

This section contains the following topics:

Topic	Page
General Information About SoMachine Templates	755
Administration of SoMachine Templates	758

General Information About SoMachine Templates

Overview

SoMachine provides templates in order to make dedicated control and visualization functionality that has been developed in one SoMachine project easily available to other SoMachine projects. They help to standardize the usage of field devices and application functions throughout different SoMachine 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

SoMachine provides various templates, but you can also create your own templates for any functionality you want to make available to other projects.

Creating Your Own Templates

The following steps are required for all SoMachine templates:

Step	Action
1	Create your functionality within a SoMachine project and test it with the appropriate hardware or in the simulation.
2	Save the functionality in a template library.
3	Open another SoMachine project and select the template from the template library in order to make the functionality available to this project.

General Notes

When using SoMachine 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.
- The templates function does not support Vijeo-Designer applications; HMI applications are not included in the SoMachine templates.
- It is possible to install one template several times on the same controller device. In order to avoid naming conflicts when creating the same objects several times, they are renamed automatically during installation. For further information, refer to the *Naming of Objects* section of the *Adding Devices from Template* chapter ([see page 770](#)).
- User-defined data types (DUT) or function blocks must be defined in a function block library if they should be used in templates.
- Templates do not support the use of direct representations of variables (for example %IX2.0).

But you can, on the other hand, use direct representations with an incomplete address specification (for example %I*). For further information, refer to the chapter *Variables configuration - VAR_CONFIG* (see page 543).

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.

SoMachine allows you to program instructions using either a direct or indirect method of parameter usage. The direct method is called Immediate Addressing where you use direct address of a parameter, such as %IWx or %QWx for example. The indirect method is called Symbolic Addressing where you first define symbols for these same parameters, and then use the symbols in association with your program instructions.

Both methods are valid and acceptable, but Symbolic Addressing offers distinct advantages, especially if you later make modifications to your configuration. When you configure I/O and other devices for your application, SoMachine automatically allocates and assigns the immediate addresses. Afterward, if you add or delete I/O or other devices from your configuration, SoMachine will account for any changes to the configuration by reallocating and reassigning the immediate addresses. This necessarily will change the assignments from what they had once been from the point of the change(s) in the configuration.

If you have already created all or part of your program using immediate addresses, you will need to account for this change in any program instructions, function blocks, etc., by modifying all the immediate addresses that have been reassigned. However, if you use symbols in place of immediate addresses in your program, this action is unnecessary. Symbols are automatically updated with their new immediate address associations provided that they are attached to the address in the I/O Mapping dialog of the corresponding Device Editor, and not simply an 'AT' declaration in the program itself.

WARNING

UNINTENDED EQUIPMENT OPERATION

Inspect and modify as necessary any immediate I/O addresses used in the application after modifying the configuration.

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

NOTE: Systematically use symbols while programming to help avoid extensive program modifications and limit the possibility of programming anomalies once a program configuration has been modified by adding or deleting I/O or other devices.

Supported I/O Modules

SoMachine templates can include the following I/O modules:

- TM2
- TM3
- TM5

Supported Fieldbusses

SoMachine 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 (LMC058)
- CANmotion

Administration of SoMachine 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

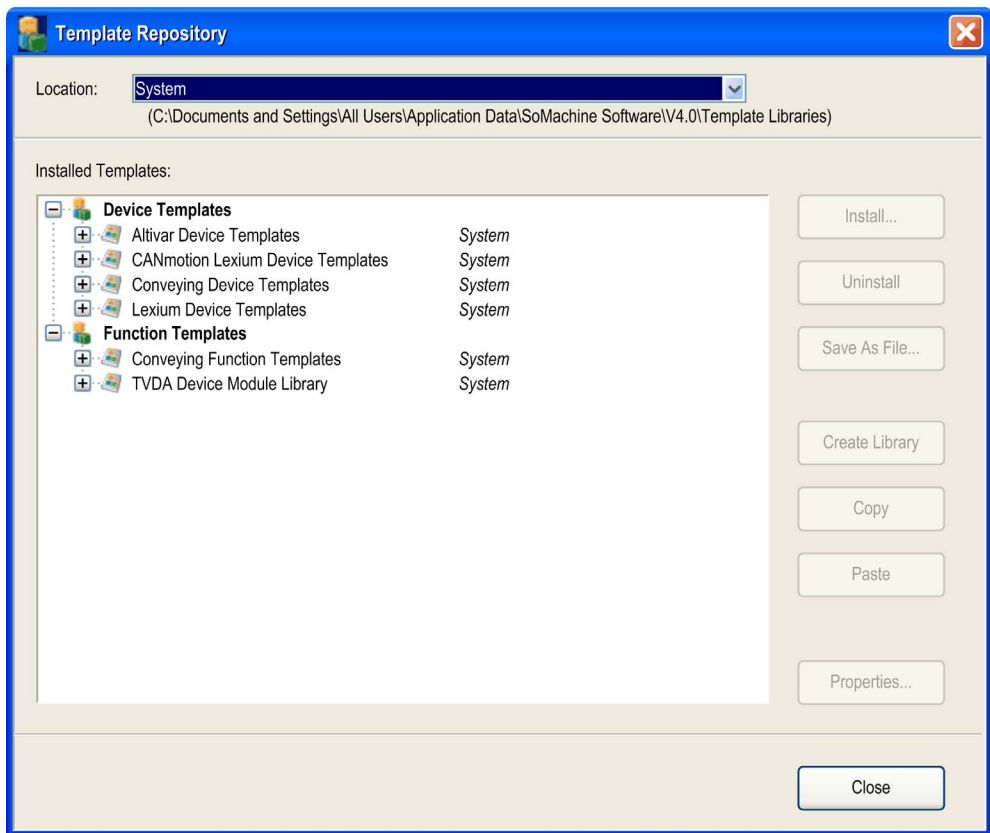
The standard template libraries included in the SoMachine scope of delivery are write-protected, which means that they cannot be deleted or renamed.

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 in SoMachine, select **Tools → Template Repository** in the SoMachine Logic Builder. To access the **Template Repository** from SoMachine Central, open the **System Options** dialog box by clicking the **System Options** button in the toolbar or by clicking the **Settings** button in the **Versions** screen. In the **System Options** dialog box, click the **Template Repository** button (*see SoMachine Central, User Guide*).

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 SoMachine 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 SoMachine

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 SoMachine-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.
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.

Properties - FUNC1

Template Name: FUNC1

Help ID: SoMProg.chm::/SoMProg_D-SE-0001286.htm#D-SE-0001286.1 Show Help

Info

Language: English Import From File

Comment: This is a short comment

Description: LONG COMMENT

Picture: PictureForATV312.jpg

13

OK Cancel

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 own 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 SoMachine. 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 (see page 764).
	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 in SoMachine.
	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>
```

Chapter 33

Managing Device Templates

Section 33.1

Managing Device Templates

What Is in This Section?

This section contains the following topics:

Topic	Page
Facts of Device Templates	767
Adding Devices from Template	768
Creating a Device Template on the Basis of Field Devices or I/O Modules	771
Visualizations Suitable for Creating Device Templates	772
Further Information on Integrating Control Logic into Device Templates	773
Steps to Create a Device Template	775

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

Already available device templates are saved 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 SoMachine 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 SoMachine 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:

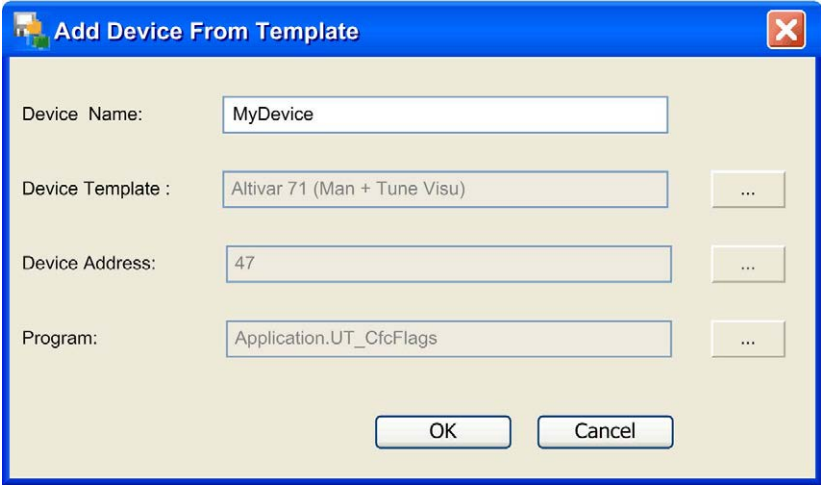
- Fieldbus device configuration
- Control logic (controller programming) (optional)
- Visualization elements (visualization programming) (optional)

You can create your own device templates from your project. For details, refer to the *Steps to Create a Device Template* chapter ([see page 775](#)).

Add Device from Template

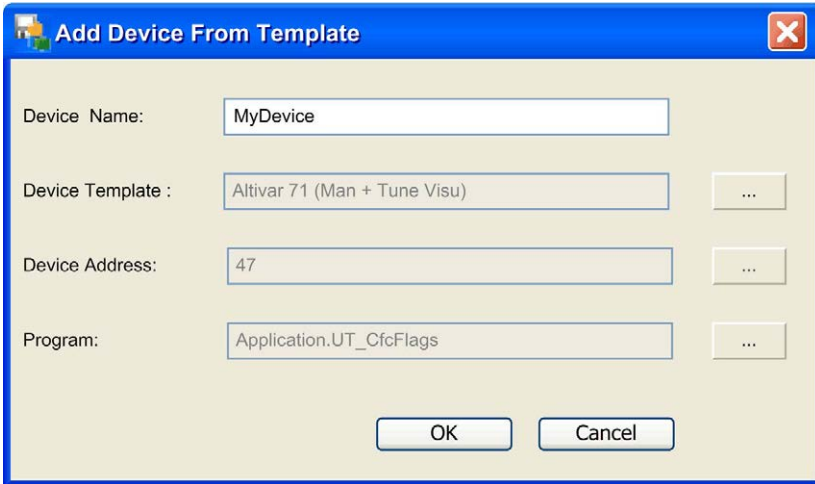
SoMachine provides 2 ways to add a device from a device template:

- 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 available in SoMachine 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 by SoMachine. Result: The Add Device From Template dialog box displays. 

Step	Action
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 context menu:

Step	Action
1	Open the Devices tree .
2	Right-click field device manager, and execute the command Add Device From Template from the context menu. Result: The Add Device From Template dialog box displays.
	
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 (*see page 757*);
- 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 (*see page 757*).

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, it is required that the control logic contains one or more code sections that exchange data with this field device. This control logic must be executed (added to a task or called by another program). Otherwise, it is not considered when executing the **Build** 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

SoMachine supports both types of visualizations:

- 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.

SoMachine references the data of the I/O device within the properties of the visual elements. When you create a new device based on this device template, SoMachine directly replaces the variables 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 **Programming with SoMachine → Visualization** of the SoMachine 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, SoMachine adapts all placeholders of the embedded visualizations 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 SoMachine can find them.

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 this 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

Only include simple control logic in a device template.

By this way, the code sections 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.

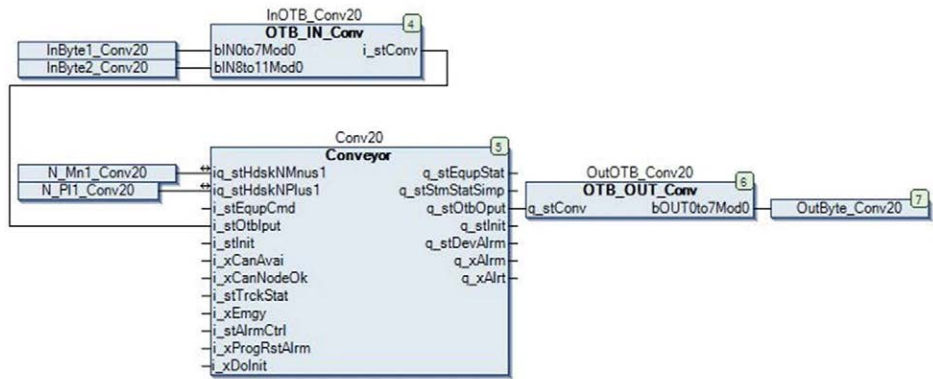
Practices for the Creation of Control Logic in CFC

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. This means that 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* (see page 771).

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 context menu. Result: SoMachine automatically builds the application. 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:

The screenshot shows the 'Save As Device Template' dialog box for an I/O device named 'Altivar_31'. The dialog has a blue title bar and a standard Windows-style window with a close button. It contains several input fields and buttons. Numbered callouts point to specific elements: 1 points to the 'Device Type' field containing 'Altivar 31'; 2 points to the 'Fieldbus Type' field containing 'CANopen'; 3 points to the 'Template Name' field containing 'Altivar 31' and a browse button (...); 4 points to the 'Template Library' field containing '<Select template library>' and a browse button (...); 5 points to a bracketed group containing the 'Function Blocks' field (containing '<Select function blocks>') and the 'Visualizations' field (containing '<Select visualizations>'), each with a browse button (...); 6 points to the 'Properties...' button at the bottom left. At the bottom right are 'OK' and 'Cancel' buttons.

- 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 (<i>see page 773</i>).
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 ([see page 762](#)).

Chapter 34

Managing Function Templates

Section 34.1

Managing Function Templates

What Is in This Section?

This section contains the following topics:

Topic	Page
Facts of Function Templates	781
Adding Functions from Template	782
Application Functions as Basis for Function Templates	788
Steps to Create a Function Template	790

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
- 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 traces
- One or several CAM tables
- 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 SoMachine 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 SoMachine 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

Procedure

SoMachine provides 2 ways to add a function from a function template:

To add an application function from a function template via drag-and-drop, proceed as follows:

Step	Action
1	Open the Macros view of the Software Catalog .
2	Select a function template from the Macros view, drag it to the Applications tree , and drop it at a suitable Application node or a folder below the Application node. Remark: Suitable nodes are highlighted by SoMachine. Result: The Add Function From Template dialog box opens.

To add an application function from a function template via context menu, proceed as follows:

Step	Action
1	Open the Applications tree .
2	Right-click an Application node or a folder below the Application node, and execute the command Add Function From Template from the context menu. Result: The Add Function From Template dialog box is displayed.

Add Function From Template Dialog Box

Add Function From Template

Function Name:

FCT1

Function Template:

FUNC

...

I/O Devices:

Device Name	Device Type	Fieldbus Type	Master	Address
FCT1_Altivar_71	Altivar 71	CANopen	CANopen_Performance	<Select device address>

I/O Mapping:

Name	Data Type	Mapping	Description
FCT1_Input1	BOOL	%IX3.1	First Input
FCT1_Input2	BOOL	%IX3.4	Second Input
FCT1_Output1	BOOL		First Output
FCT1_Output2	BOOL		Second Output

Parameters:

Object	Name	Data Type	Default	New Value	Description
FCT1_POU	InternalVar1	STRING	'XXXX'		Internal Variable1
FCT1_POU	InternalVar2	INT	66		Internal Var2
FCT1_POU	ControlWord1	INT	66		ControlWord1 : Just a variable

OK

Cancel

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.

Element		Description
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.
	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: SoMachine verifies whether the settings are correct and inserts the new application function as separate node below the Application node or displays an error detection message.

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.

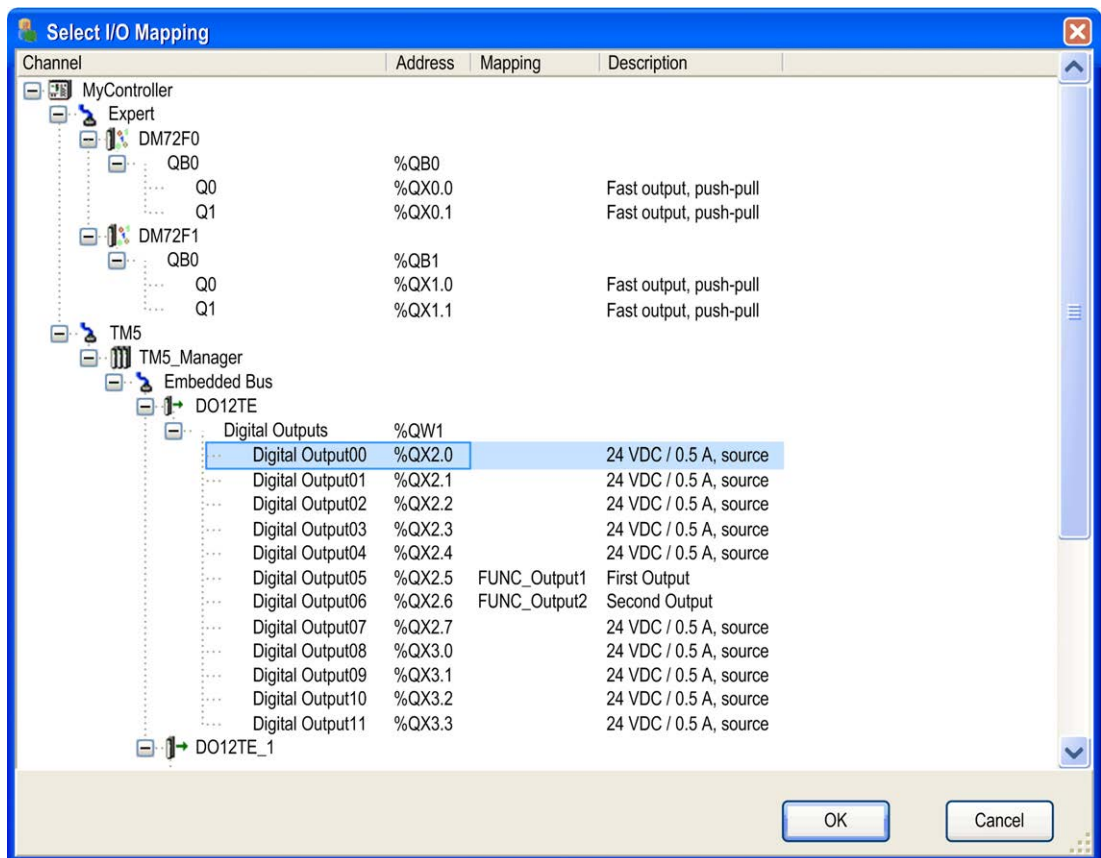
2 data types are compatible if they have identical type names or if they are elementary IEC data types of the same size.

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 view 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.
Visualizations	The visualizations that are included in the function template are created below the root folder using names that apply to the naming rules. The properties of the visualization are automatically adjusted.
Programs	The programs that are included in the function template are created below the root folder using names that apply to the naming rules. The names of those objects in the program that are part of the function template is adjusted automatically.
Traces	The traces that are included in the function template are created below the root folder using names that apply to the naming rules and can be used to trace variables belonging to the application function.
CAM tables	The CAM tables that are included in the function template are created below the root folder using names that apply to the naming rules. They are only required if the application function includes SoftMotion devices.
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.

Object	Description
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 not correct, SoMachine issues an error detection message.
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 not correct, SoMachine issues a message.

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.

Defining Application Functions as Function Templates

You can save application functions as function templates by right-clicking a subnode of your **Application** node in the **Applications tree**. Or you can create your own template in the **Macros** view by selecting individual objects for your template. These 2 procedures are described in the *Steps to Create a Function Template* chapter ([see page 790](#)).

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 **Build** 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 ([see page 783](#)).

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 ([see page 145](#)), 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 ([see page 794](#)).

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 ([see page 783](#)).

Objects Saved in Function Templates

The following objects are saved in function templates:

- All programs that are located directly in the application function folder as well as their subobjects
- All global variable lists that are located directly in the application function folder
- All visualizations that are located directly in the application function folder
- All CAM tables that are located directly in the application function folder
- All traces that are located directly in the application function folder
- All field devices and I/O modules that are used by any program or visualization that is included in the function template
- All global variables whose variable lists are not part of the function template but which are used by any program or visualization that is part of the function template
- All persistent variables that are used by any program or visualization that is part of the function template

NOTE: Any other object types are not saved in the function template (even if they are saved in the application function folder). Only use function blocks and data types that are stored in a library.

Steps to Create a Function Template

Overview

SoMachine provides 2 ways to create a function template:

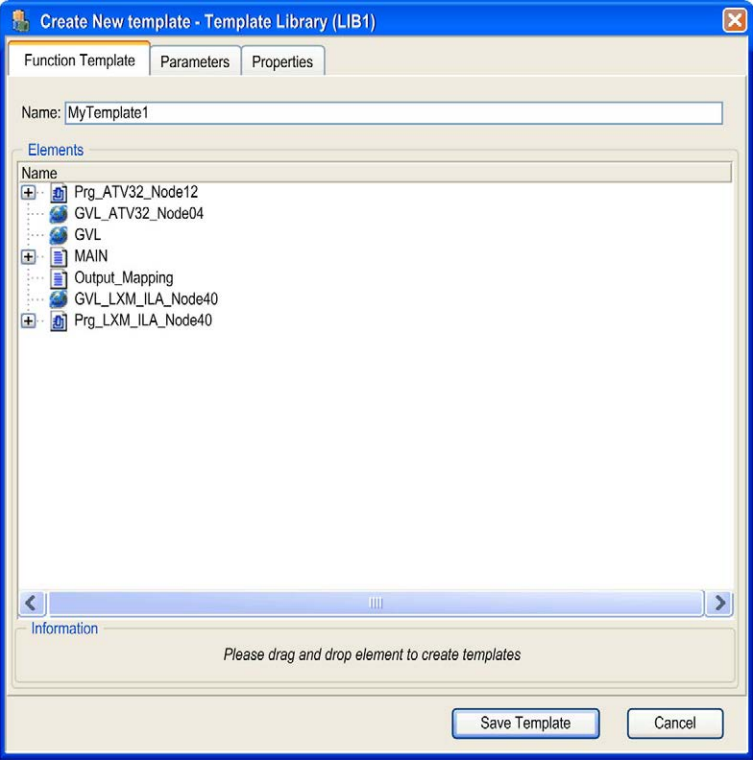
- From the **Macros** view using the **Create New Template** dialog box.
- From the **Applications tree** using the **Save as Function Template** dialog box.

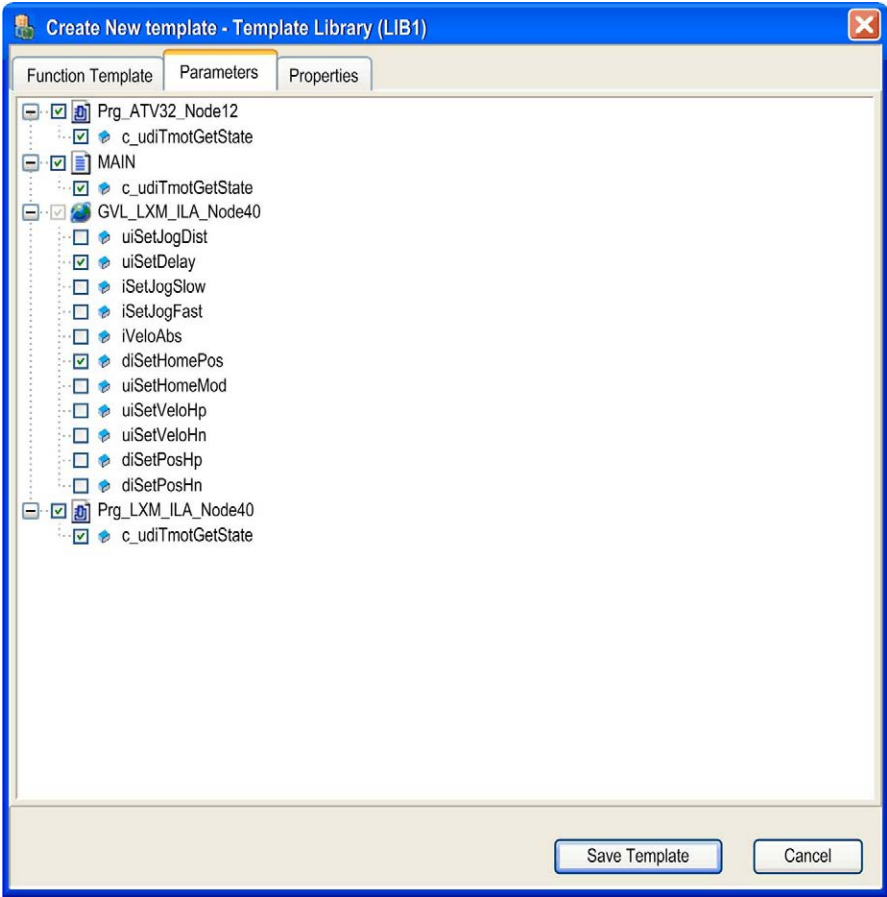
The following paragraphs list the steps that have to be performed in order to save already available application functions that meet the criteria stated in *Application Functions as Basis for Function Templates* ([see page 788](#)) as function templates.

Procedure via Macros View

The procedure via **Macros** view allows you to create your own function template by dragging and dropping elements:

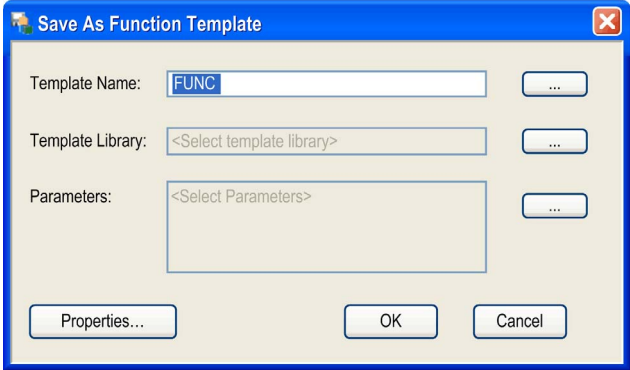
Step	Action
1	In the Macros view, expand the section My Template .
2	Select the My Template node, and click the green plus button. Result: A new node with the default name LIB1 is inserted below the MyTemplate node.

Step	Action
3	Select the LIB1 node and click the green plus button. Result: The Create New Template dialog box displays. 
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.

Step	Action
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.  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 <i>Adding Further Information for Templates or Template Libraries</i> (see page 762).
7	Click the Save Template button.

Procedure via Applications tree

To save an already available application function as function template, proceed as follows:

Step	Action
1	Right-click a subfolder of your Application node in the Applications tree .
2	Select the command Save As Function Template from the context menu. Result: SoMachine automatically builds the application. After the built process has been successfully completed, the Save As Function Template dialog box will be 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: SoMachine verifies that the function template can be created and displays a message that the function template has been created successfully or indicates the errors detected.

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.

Step	Action
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 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 (*see page 789*).

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 Values 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 (see page 762)*.

Part IX

Troubleshooting and FAQ

What Is in This Part?

This part contains the following chapters:

Chapter	Chapter Name	Page
35	Generic - Troubleshooting and FAQ	797
36	Accessing Controllers - Troubleshooting and FAQ	807

Chapter 35

Generic - Troubleshooting and FAQ

Section 35.1

Frequently Asked Questions

What Is in This Section?

This section contains the following topics:

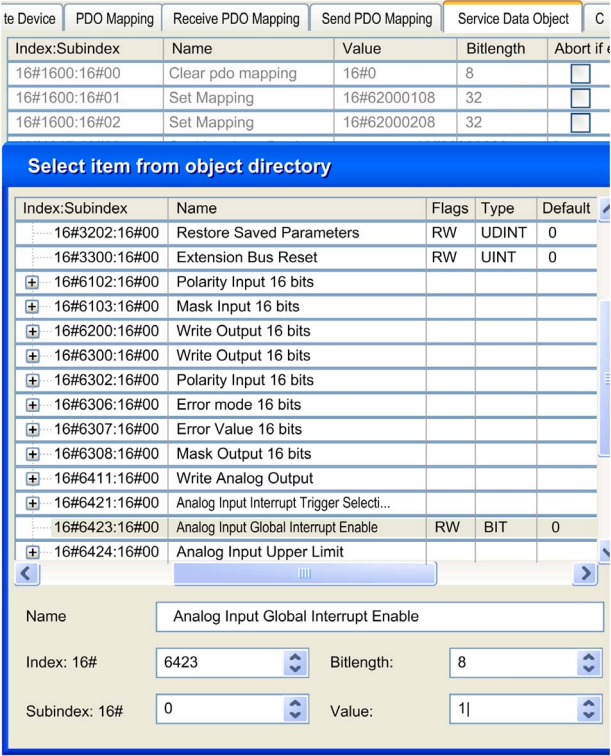
Topic	Page
How Can I Enable and Configure Analog Inputs on CANopen?	799
Why is SoMachine Startup Performance Sometimes Slower?	800
How Can I Manage Shortcuts and Menus?	801
How Can I Increase the Memory Limit Available for SoMachine on 32-Bit Operating Systems?	803
How Can I Reduce the Memory Consumption of SoMachine?	804
How Can I Increase the Build-Time Performance of SoMachine?	804
What Can I Do in Case of Issues with Modbus IOScanner on Serial Line?	805
What Can I Do If My Network Variables List (NVL) Communication Has Been Suspended?	806
What Can I Do If a Multiple Download is Unsuccessful on an HMI Controller?	806

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.
3	Open the Service Data Object tab and click the New... button. Result: The Select item from object directory dialog box is displayed.  The screenshot shows the 'Service Data Object' tab in the CANopen Remote Device editor. It contains a table with columns: Index:Subindex, Name, Value, Bitlength, and Abort if error. The table lists several SDOs, including 'Clear pdo mapping', 'Set Mapping', and 'Set Mapping'. Below the table is a 'Select item from object directory' dialog box. This dialog box has a table with columns: Index:Subindex, Name, Flags, Type, and Default. It lists various SDOs, including 'Restore Saved Parameters', 'Extension Bus Reset', 'Polarity Input 16 bits', 'Mask Input 16 bits', 'Write Output 16 bits', 'Error mode 16 bits', 'Error Value 16 bits', 'Mask Output 16 bits', 'Write Analog Output', 'Analog Input Interrupt Trigger Select...', 'Analog Input Global Interrupt Enable', and 'Analog Input Upper Limit'. The 'Analog Input Global Interrupt Enable' SDO is selected. Below the table, there are input fields for Name, Index: 16#, Subindex: 16#, Bitlength, and Value. The Name field is set to 'Analog Input Global Interrupt Enable', Index: 16# is set to 6423, Subindex: 16# is set to 0, Bitlength is set to 8, and Value is set to 1.

Step	Action
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 SoMachine Startup Performance Sometimes Slower?

Overview

Beside the PC configuration there are several other conditions which can increase the time SoMachine is consuming during startup:

Boot Phase	Startup Performance
first start after SoMachine installation	On first start after SoMachine has been installed, 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 of SoMachine can be longer than usual because Microsoft Windows consumes some time in the background to launch services that are needed to run SoMachine. 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 SoMachine software differ depending on the current state, that is, the window or editor that is currently open.

You can adapt the shortcuts and menus to your individual preferences or you can load the SoMachine 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 SoMachine Standard Shortcuts and Menus

To restore the SoMachine 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 <i>... Program Files Schneider Electric SoMachine Software V4.0 LogicBuilder Settings</i> , select the file <i>Standard.opt.menu</i> , and click Open . Result: The Customize dialog box now shows the standard SoMachine settings.
4	To load these standard settings to the SoMachine graphical user interface, click OK .

Setting the Shortcuts and Menus to CoDeSys Standard

To import the CoDeSys shortcuts and menus to your SoMachine 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 <i>... Program Files Schneider Electric SoMachine Software V4.0 LogicBuilder Settings Original CoDeSys</i> , select the file <i>Standard.opt.menu</i> , and click Open . Result: The Customize dialog box now shows the CoDeSys settings.
4	To load these CoDeSys settings to the SoMachine graphical user interface, click OK .

NOTE: The menus and shortcuts of the SoMachine software differ, depending on the window or editor that is currently open.

Expanding Menus

SoMachine main menus and context 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 Memory Limit Available for SoMachine on 32-Bit Operating Systems?

Overview

Large SoMachine projects can stress a 32-bit operating system to the technical limit regarding memory consumption. This is due to 32-bit operating systems providing only 2 GB of memory for user processes such as SoMachine.

Identifying a Large SoMachine Project

SoMachine projects can be considered to be large if they contain a large total count of **Objects**. The **Objects** that are available in a project, such as **Devices**, **POUs**, **Actions**, **DUTs**, **Global Variable Lists**, **Visualizations** are listed in the **Statistics** tab of the **Project Information** dialog box (see *SoMachine, Menu Commands, Online Help*). However, it is not just the total count of **Objects** that can indicate a large project. Even individual **Objects** can be inherently large.

Enabling the 3 GB Switch on Windows 7 32-Bit Operating Systems

In order to increase the memory limit for user processes on Windows 7 32-bit Operating Systems, you can enable the 3 GB switch function as follows:

Step	Action
1	Goto Start Menu → All Programs → Accessories .
2	Right-click Command Prompt and execute the command Run as Administrator .
3	Enter <code>bcdedit /set IncreaseUserVa 3072</code> .
4	Restart the computer.

Disabling the 3 GB Switch on Windows 7 32-Bit Operating Systems

In order to increase the memory limit for user processes on Windows 7 32-bit Operating Systems, you can enable the 3 GB switch function as follows:

Step	Action
1	Goto Start Menu → All Programs → Accessories .
2	Right-click Command Prompt and execute the command Run as Administrator .
3	Enter <code>bcdedit /deletevalue IncreaseUserVa</code> .
4	Restart the computer.

How Can I Reduce the Memory Consumption of SoMachine?

Overview

This chapter provides tips that may help to reduce the memory consumption of the SoMachine process on your system.

Splitting up Your SoMachine Project

If your SoMachine project consists of several independent parts, for example, independent root devices (controller or panel), you can split up the project and create independent SoMachine projects for each root device. These smaller SoMachine projects then require less memory each.

Closing SoMachine Editors

Close each SoMachine editor after you have made the respective settings because every open editor consumes memory space.

How Can I Increase the Build-Time Performance of SoMachine?

Best Practices

The list of best practices may help you to avoid a slow performance when working with SoMachine:

- Verify that the hardware of the PC meets the system requirements (*see SoMachine, Introduction*).
- 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 via the SoMachine Configuration Manager.
- If you are using Vijeo-Designer integrated in SoMachine, consider to disable the automatic symbol export function. To achieve this, activate the option **Disable automatic symbol export** in the SoMachine Logic Builder **Options → Vijeo-Designer** dialog box (*see page 500*).
- Consider activating the option **Disable undo after deleting a DTM (performance optimization)** in the SoMachine Logic Builder **Options → FDT Options** dialog box (*see SoMachine, Menu Commands, Online Help*) if this function is not frequently used.
- Avoid large POUs. SoMachine has no limit for the size of POUs programmed in graphical languages like LD. Nevertheless, it is a best 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.

What Can I Do If a Multiple Download is Unsuccessful on an HMI Controller?

Problem

A multiple download is not successfully completed on an HMI controller with an outdated firmware.

Solution

Restart the multiple download or execute the **Runtime Installer** in SoMachine Central via **Tool Access Bar → Maintenance → Download Firmware HMI**.

Chapter 36

Accessing Controllers - Troubleshooting and FAQ

What Is in This Chapter?

This chapter contains the following sections:

Section	Topic	Page
36.1	Troubleshooting: Accessing New Controllers	808
36.2	FAQ - What Can I Do in Case of Connection Problems With the Controller?	813

Section 36.1

Troubleshooting: Accessing New Controllers

What Is in This Section?

This section contains the following topics:

Topic	Page
Accessing New Controllers	809
Connecting via IP Address and Address Information	811

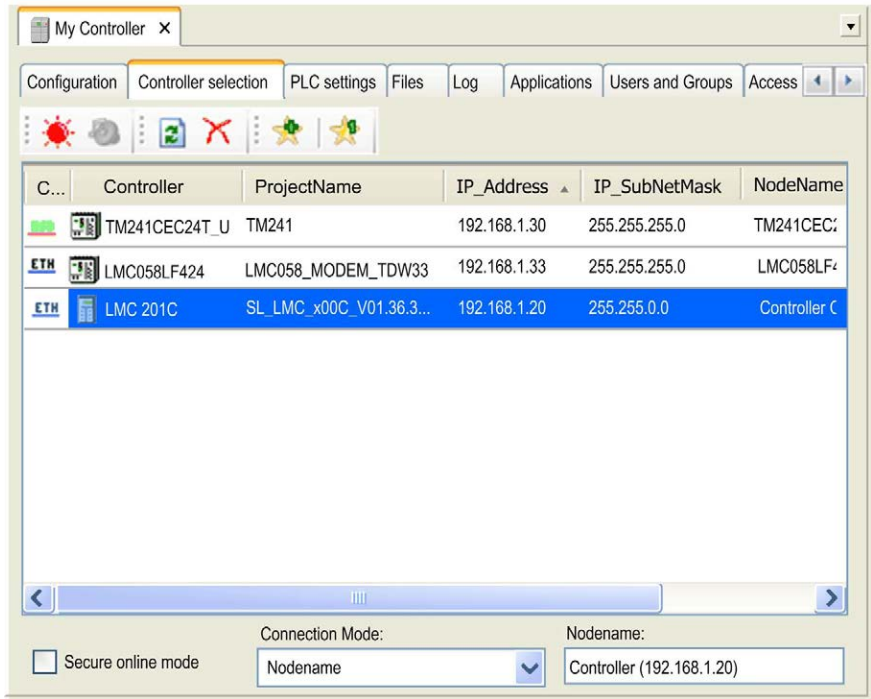
Accessing New Controllers

Overview

To access a new controller, adapt the network settings of the controller to the network of your SoMachine PC. This chapter provides a step-by-step example.

Example

This example shows the steps to access an LMC058 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 running SoMachine or to the network of the PC using an Ethernet cable.
2	In SoMachine, open the Controller selection view of the device editor (<i>see page 99</i>). Result: The LMC058 controller will be included in the list. 

Step	Action
3	To adapt the communication settings of the controller, right-click the controller in the Controller selection list, and execute the command Process communication settings... from the context menu. Result: The Process communication settings dialog box opens.
4	In the Process 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 Process communication settings dialog box.
6	In the Controller selection 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.

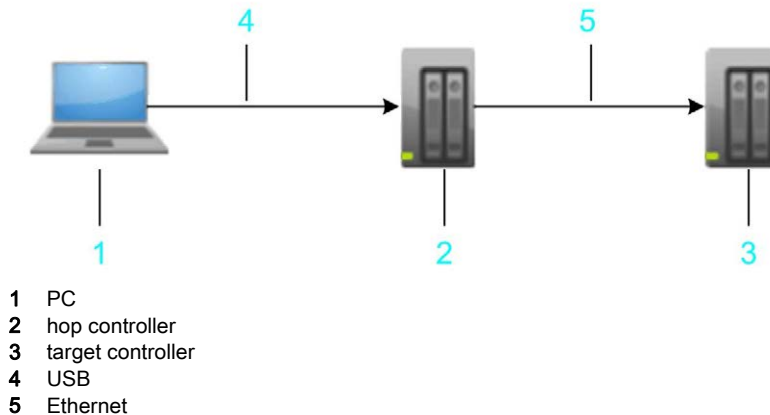
Failure to follow these instructions can result in death, serious injury, or equipment damage.

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

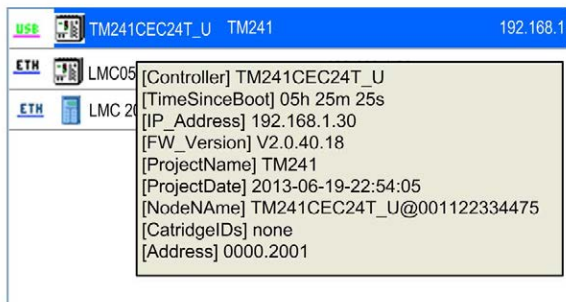
The used communication protocol offers a mechanism to connect to a controller independent of the type of connection. For example, this allows access to a target controller that is connected via Ethernet to another hop controller that is connected via USB to the PC itself.



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 Controller Selection View? ([see page 815](#))) 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)**.

Section 36.2

FAQ - What Can I Do in Case of Connection Problems With the Controller?

What Is in This Section?

This section contains the following topics:

Topic	Page
FAQ - Why is a Connection to the Controller not Possible?	814
FAQ - Why has the Communication Between PC and Controller been Interrupted?	817

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 (*see page 809*), and you still cannot connect to the controller, the reason can be the subnet mask. Since the used 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	In SoMachine, open the Controller selection view of the device editor (<i>see page 99</i>).
2	To adapt the communication settings of the controller, right-click the controller in the Controller selection list, and execute the command Process communication settings... from the context menu. Result: The Process communication settings dialog box opens.
3	Adapt the Subnet mask configured for the controller exactly to the subnet mask of your SoMachine PC. Example: Change 255.255.0.0 to 255.255.255.0 .

NOTE: After you have changed the **Connection Mode** in the **Controller selection** 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 SoMachine Central, SoMachine Logic Builder, Controller Assistant) and a controller, a running SoMachine gateway is required. If you attempt to login to a controller, the application automatically starts the active SoMachine gateway. If SoMachine 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 context 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 View?

If you establish a connection between the controller and the SoMachine PC by using the active path, then the **Communication Settings** view is displayed in the device editor (*see page 114*). This is the default setting for SoMachine V3.1 and earlier versions.

If the controller of your choice is not displayed in the **Communication Settings** view, you can temporarily switch to connection establishment via IP address as follows:

Step	Action
1	Open the Project Settings → Communication settings dialog box from the Project menu.
2	Select the option Dial up via "IP-Address" and confirm your setting by clicking OK .
3	Adapt the network settings of the controller to the network of your SoMachine PC as described in the <i>Accessing New Controllers</i> chapter (<i>see page 809</i>).
4	Set connection establishment back to active path. The controller should now be displayed in the Communication Settings view.

Why is the Controller not Listed in the Controller Selection View?

If you do not find your controller in the list of the **Controller selection** view of the device editor (*see page 99*), the reason can be that 2 different devices are assigned the same **Nodename**. If 2 devices are assigned the same **Nodename**, only 1 of these devices is listed in the **Controller selection** 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	Right-click the device that is assigned a double Nodename in the Controller selection list and execute the command Change device name from the context menu. Result: The Change device name dialog box is displayed. The image shows a 'Change device name' dialog box. It has a blue title bar with a red 'X' button. Inside, there is a yellow warning triangle icon. The text says: 'The device name must be unique in the current network! Otherwise it won't be listed correctly and connecting can lead to unexpected behaviour.' Below this, there is a 'Device name' label and a text box. The 'Current' name is 'TM241CEC24T_U @001122334475'. The 'New' name is 'MyDevice (192.168.1.30)'. At the bottom, there are 'OK' and 'Cancel' buttons.
2	In the Change device name dialog box, enter a unique Nodename in the New text box.
3	Click OK to confirm and to close the Change device name dialog box.
4	In the Controller selection view, click the Update 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.
5	Repeat steps 1...4 until you have eliminated any double Nodenames .

NOTE: Some controllers, such as the LMC •0•C controllers, create a **Nodename** automatically out of the device name of the project after a project download and the IP address (for example, **MyLMC (192.168.1.30)**). This automatic name overwrites the **Nodename** you assigned if any changes are executed on the controller.

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 context menu.

Appendices



What Is in This Appendix?

The appendix contains the following chapters:

Chapter	Chapter Name	Page
A	Network Communication	821
B	Usage of the OPC Server 3	829
C	Script Language	843
D	User Management for Soft PLC	889
E	Controller Feature Sets for Migration	899

Appendix A

Network Communication

What Is in This Chapter?

This chapter contains the following topics:

Topic	Page
Network Topology	822
Addressing and Routing	823
Structure of Addresses	825

Network Topology

Overview

The SoMachine 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 is simple enough that any node in the network, that is, even nodes with low resources, are able to 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 single parent only.

Addressing and Routing

Overview

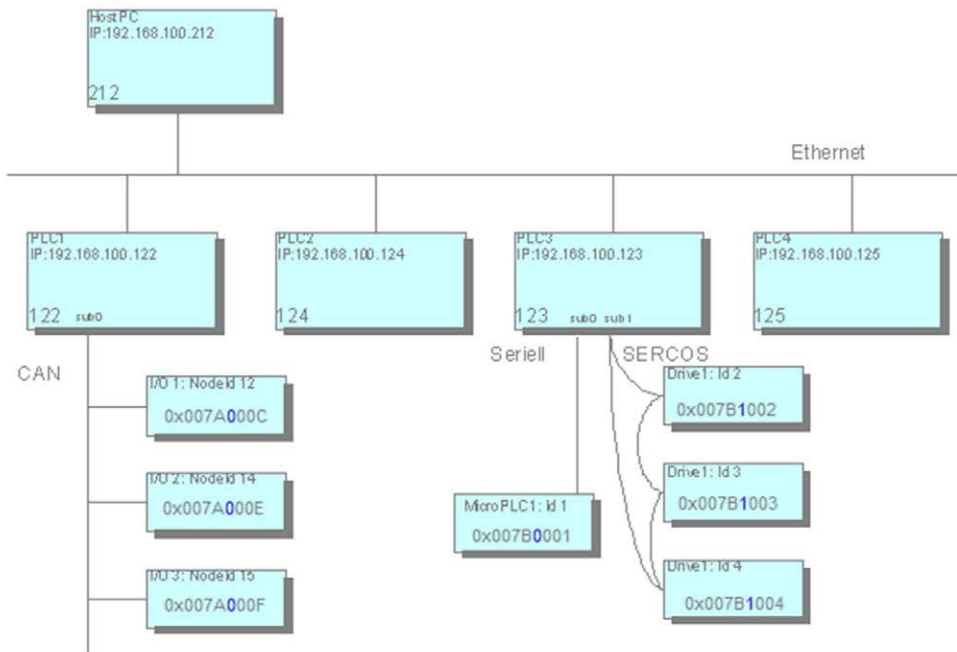
Addressing maps the topology of the control network to unique addresses. A node address (*see page 826*) is built up hierarchically.

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 (*see page 827*) 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 (*see page 825*)
- Node Addresses (*see page 826*)
- Absolute and Relative Addresses (*see page 827*)
- Broadcast Addresses (*see page 828*)

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: $b = (n + 7) \text{ DIV } 8$
- The $(n \text{ MOD } 8)$ lowest-order bits of the first byte and all remaining $(n \text{ DIV } 8)$ bytes are used for the network address.

Example - Network Address

Length: 11 bit

Address: 111 1000 1100

Example for network address coding

Byte	0								1							
Bit	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
						1	1	1	1	0	0	0	1	1	0	0
	Reserved (0)															

Node Addresses

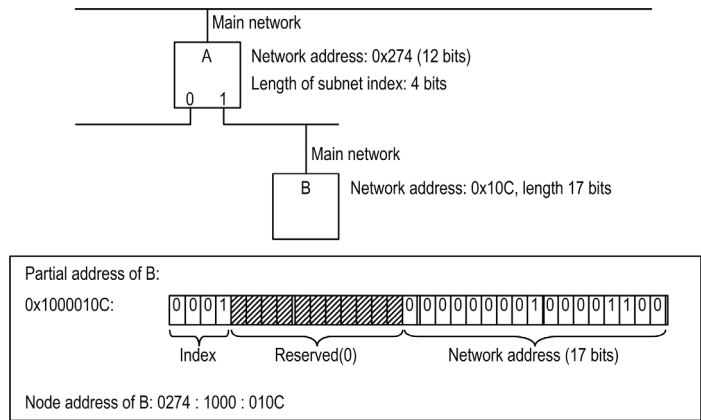
The node address indicates the absolute address of a node within a control network, and therefore, is unique within the whole tree. The address consists of up to 15 address components, each consisting of 2 bytes. The lower a node is located within the network hierarchy, the longer its address.

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: This means that 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. This means that 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 $1 \geq 0$, 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.

Appendix B

Usage of the OPC Server 3

Overview

The description provided in this chapter is dedicated to persons experienced in OPC server technology.

The file *OPC_V3_how_to_use_E.pdf*, that is automatically installed with SoMachine in the directory *C:\Program Files\Schneider Electric\SoMachine OPCServer* provides a more detailed description of how to configure the OPC server.

What Is in This Chapter?

This chapter contains the following topics:

Topic	Page
General Information	830
Declaring a Variable to be Used With OPC	832
OPC Server Configuration	836
Usage of the CoDeSys OPC Server	842

General Information

What is OPC?

OPC is a standardized interface for access to process data. It is based on the Microsoft standard COM/DCOM2 (Component Object Model / Distributed COM) that has been extended due to the requirements of data access in automation, where the interface is mainly used to read data from or write data to the controller.

Examples of typical OPC clients

- visualizations
- programs whose purpose is to collect operating data

Examples of typical providers of OPC servers

- controller systems
- field bus interface cards

What is an OPC Server?

The OPC server is an executable program that is started automatically during the establishment of a connection between client and controller. Hence, the OPC server is able to inform the client about changed variable values or states.

The OPC server provides all variables (referred to as **Items** in OPC) that are available on the controller (**Item Pool** or **Address Space**). These items are managed within a **Data Cache** that helps to ensure fast access to their values. Also possible is a direct, not cached access to the items of the controller.

In the OPC server the items can be organized in so-called **Groups (Private and Public)**.

The private groups can be composed in the client arbitrarily from particular items. Initially they do not effect the groupings in the OPC server, but if necessary can be transformed to Public Groups. Working with Private Groups, for example, is useful in order to be able to activate or deactivate certain groups of variables with just one single command, depending on whether they should be accessible or not.

Grouped data should be read from the OPC server coherently, that is, all variables should be read at the same time. However, this is not always possible in case of target systems with restricted communication buffers.

Due to the characteristics of COM / DCOM it is possible to access an OPC running on another computer. It is also allowed that more than one client accesses the data source at the same time. The applicability of different languages (C++, Visual Basic, Delphi, Java) is another benefit of the concept.

Overview of the CoDeSys OPC Server 3

The CoDeSys OPC server is based on the PLCHandler of 3S - Smart Software Solutions GmbH. This communication module permits a direct communication to those controllers that are programmable with CoDeSys.

The OPC server V.3 or later supports the following OPC specifications:

- OPC Common Definitions and Interfaces Version 1.0
- Data Access Custom Interface Standard Version 1.0
- Data Access Custom Interface Standard Version 2.05A
- Data Access Custom Interface Standard Version 3.0
- Data Access Automation Interface Standard Version 2.0

Communication between OPC server and controller can be carried out via the following interface:

- Gateway V3 (parameter **Interface** → **GATEWAY3** in the OPC configuration tool (*see page 838*))

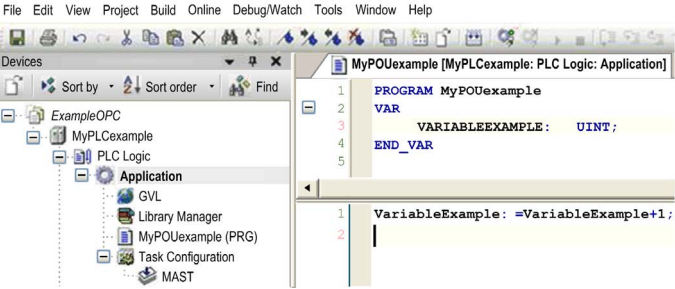
NOTE: You can configure the OPC server for simulation with the parameter **Interface** → **SIMULATION3**, but the values of the configured variables cannot be read or written.

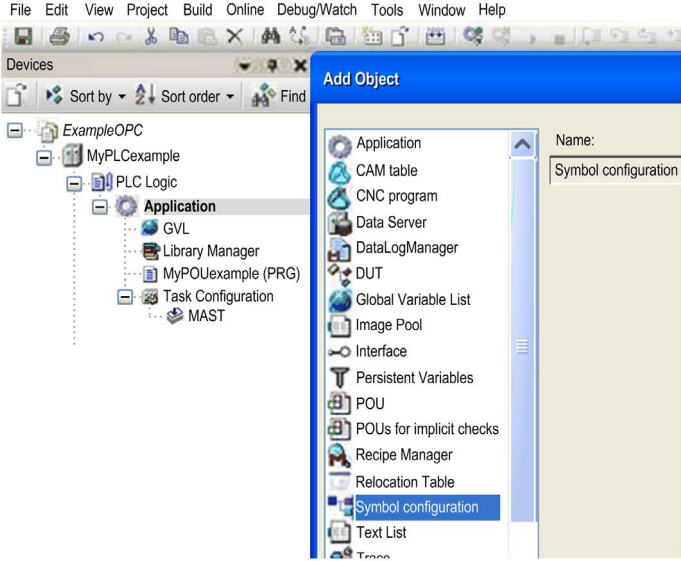
Declaring a Variable to be Used With OPC

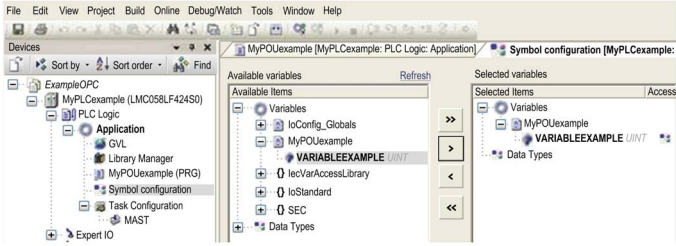
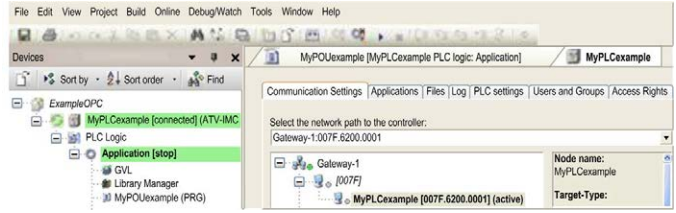
Steps to Declare a Variable

Declare a variable to be used with OPC as follows:

Step	Action	Example
1	Create a project.	ExampleOPC
2	Add and select a controller using the Add Device command.	–
3	Configure the name of the device by clicking the device node twice to make it editable.	MyPLCexample
4	Create a PROGRAM in your application by right-clicking the Application item and executing the command Add Object → POU...	Example program: Increment a UINT variable: VARIABLEEXAMPLE
5	Configure the name of the PROGRAM .	MyPOUexample
6	Double-click a task and associate the PROGRAM to the task.	Task example: MAST PROGRAM example: MyPOUexample
7	Execute the Build All command from the Build menu and verify there is no error detected during execution of the Build command.	–



Step	Action	Example
8	Create a Symbol configuration object in your application by right-clicking the Application item and executing the command Add Object → Symbol configuration.... 	–
9	In the Add Symbol configuration dialog box click Open .	–
10	Click the Refresh link.	–
11	Expand the Variables item in the Available variables list.	–
12	Select the variable you want to share with your OPC client from your program.	Variable example: VARIABLEEXAMPLE Program example: MyPOUexample

Step	Action	Example
13	Click the > button to send the variable to a shared data base to make it accessible for the OPC client. 	–
14	Execute the Build All command from the Build menu and verify there is no error detected during execution of the Build command.	–
15	Close the tab Symbol configuration.	Remark: In the directory where you have stored your project you will find an XML file which includes a list of the variables which are accessible for the OPC client.
16	Connect your PC to your controller by using the Communication Setting tab.	–
17	Download the application.	–
18	Start the application. 	–

XML File Listing the Variables Accessible for the OPC Client

In the directory where you have stored your project, an XML file is automatically created which describes the list of the variables which are accessible for the OPC client.

Variables Mapped to %I

Variables mapped to %I selected in a **Symbol configuration** are not automatically available inside an OPC client.

Add the path to the variable manually.

Example:

```
M251.Application.IOCONFIG_GLOBALS_MAPPING.temperatureLabo
```

OPC Server Configuration

Starting the OPC Configuration Tool

Configure the OPC server and link it with the project you have created, as follows:

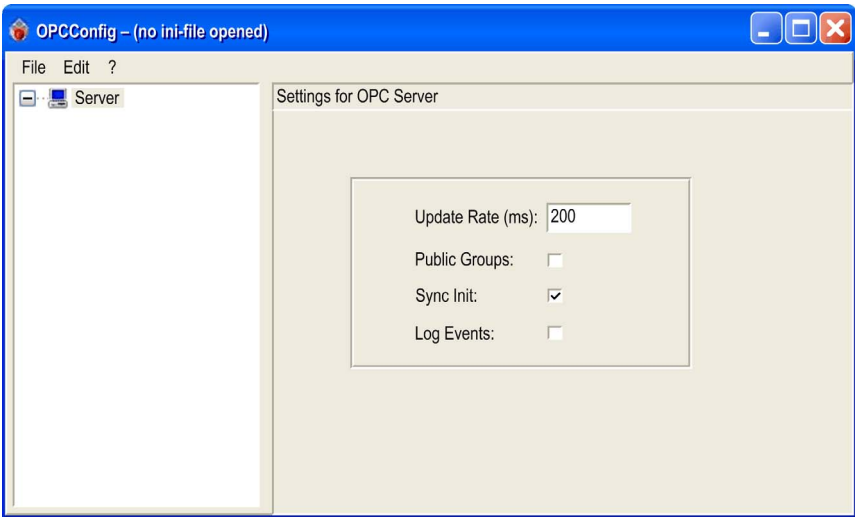
Step	Action
1	Go to the directory: <i>C:\Program Files\Schneider Electric\SoMachine OPCServer</i>
2	Double-click the: <i>OPCConfig.exe</i> file
3	The configuration tool <i>OPCconfig.exe</i> allows to generate an INI file which is needed to initialize the OPC server with the desired parameters for the communication between the CoDeSys project and the controller(s).

OPC Configuration Tool

The configuration tool contains the following elements:

- a menu bar
- a tree view for mapping the assignments of one or several controllers to the server
- a configuration dialog that corresponds to the currently selected tree entry

After having started the tool, it will appear as follows, containing the default common settings:



File Menu of the OPC Configuration Tool

The **File** menu provides commands for loading and saving the configuration files to / from the configuration tool:

Command	Shortcuts	Description
Open	CTRL+O	For editing an existing configuration. The default dialog box for opening a file opens. Select an already existing INI file. The filter is automatically set to <i>OPCconfig Files *.ini</i> . The configuration described in the chosen INI file will be loaded to the configuration tool.
New	CTRL+N	For creating a new configuration. If a configuration is open, you will be asked whether it should be saved before being closed. Then the configuration tool will show the default settings.
Save	CTRL+S	Saves the current configuration to the currently loaded INI file.
Save as	–	Saves the current configuration to a file by another name that you can specify in the default dialog box.
<n> recently opened INI-files	–	List of the INI files which have been edited since having last started the tool. You can select a file to get it reloaded in the configuration tool.
Exit	–	Terminates the tool. If any changes to the current configuration have not been saved, you will be asked to do so.

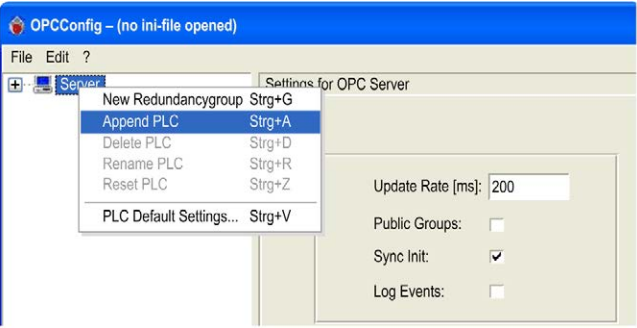
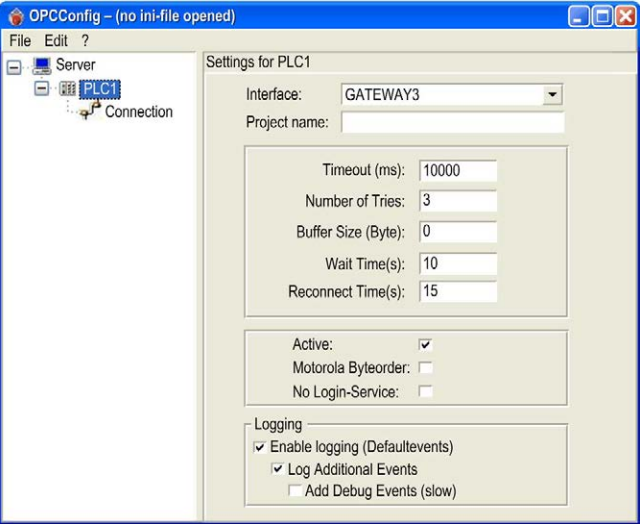
Edit Menu of the OPC Configuration Tool

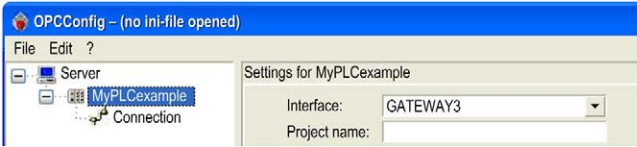
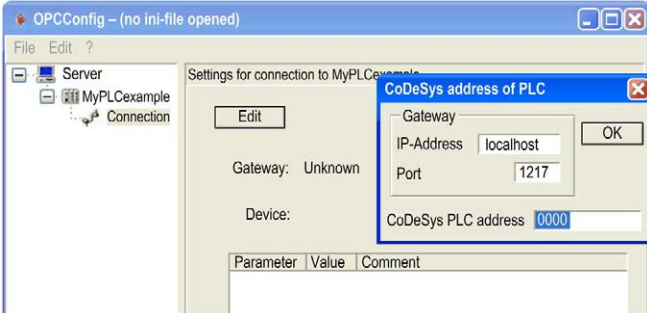
The **Edit** menu provides the commands for editing the configuration tree in the left part of the configurator.

Command	Shortcuts	Description
New Redundancygroup	CTRL+G	A Redundant Group entry will be added below Server . If there are already controller or Redundant Groups listed in the tree, the new Redundant Group will be appended at the end. By default, a new entry is named Redundant<n> , with n being a consecutive number starting with 1. To rename the entry, select it in the tree and either use the command Edit → Rename PLC or click it twice to make it editable.
Append PLC	CTRL+O	A controller entry will be added below Server . A new controller will be appended at the end of the existing tree. By default, a new entry is named PLC<n> , with n being a consecutive number starting with 1. To rename the entry, select it in the tree and either use the command Edit → Rename PLC or click it twice to make it editable.
Delete PLC	CTRL+D	The currently selected controller entry will be removed from the configuration tree.
Rename PLC	CTRL+R	The currently selected controller entry can be renamed.
Reset PLC	CTRL+Z	The settings of the currently selected controller entry will be reset to the default values defined in the PLC Default Settings .
PLC Default Settings...	–	not yet available

Configuring the OPC Server

Configure the OPC server as follows:

Step	Action
1	Right-click the Server icon and execute the Append PLC command: 
2	Select GATEWAY3 from the Interface list. 

Step	Action
3	Double-click the PLC1 item to rename it (for example: MyPLCexample). 
4	Double-click the Connection icon and click the Edit button. Result: The CoDeSys address of PLC dialog box will be displayed: 
5	To be able to access your variable from your OPC client, enter the address of the controller (for example: MyPLCexample). The address is given in the SoMachine Communication Setting dialog box of your project.

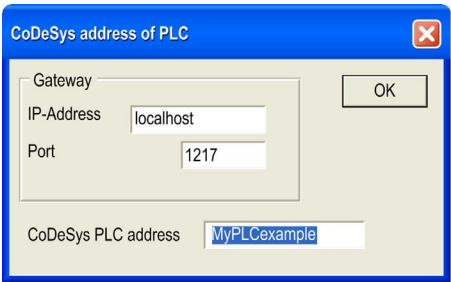
The address can be physical or logical. To avoid address value reconfigurations when there are many devices in your project, you should use logical addresses.

Logical Addressing

In our example the address is: **MyPLCexample**.

Enter directly the **Node name** given in the **Communication Setting** tab of **MyPLCexample** in your project. To configure the **Node name**, click the **Edit** button.





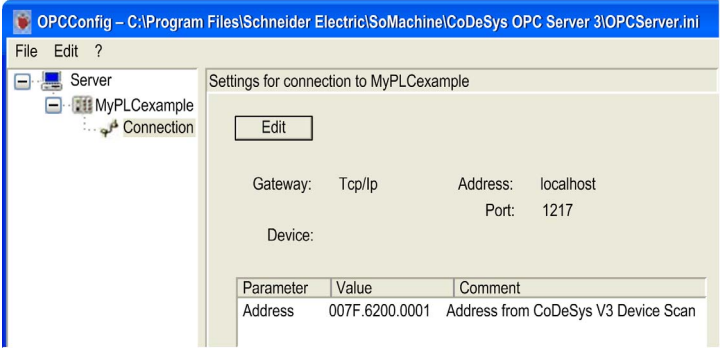
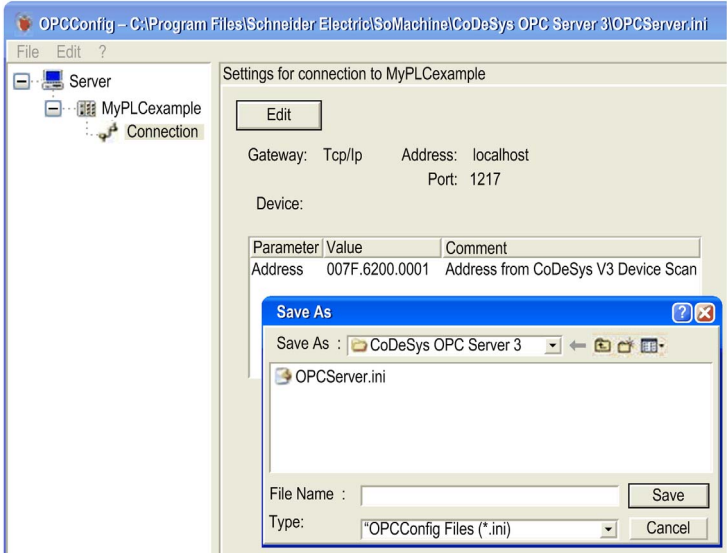
Physical Addressing

In our example the address is: 007F.6200.0001



Proceed as follows:

Step	Action
1	Take this address and configure the value of the CoDeSys address of PLC : A screenshot of the OPCConfig application window showing the configuration for the CoDeSys OPC Server. The "Settings for connection to MyPLCexample" dialog is open, displaying "Gateway: Tcp/Ip" and "Device: CoDeSys PLC address 007F.6200.0001". The "CoDeSys address of PLC" dialog is also open, showing "IP-Address: localhost", "Port: 1217", and "CoDeSys PLC address: 007F.6200.0001". The background shows the OPCConfig interface with a tree view containing "Server", "MyPLCexample", and "Connection".

Step	Action
2	Click the OK button. Result: The following dialog box will be displayed: 
3	Open the menu File and execute the command Save As. Click the OK button and the following dialog box will be displayed: 
4	Select the <i>OPCServer.ini</i> and click Save. NOTE: Please note that the name of the file must be <i>OPCServer.ini</i>. Do not use another file name.

Usage of the CoDeSys OPC Server

Overview

After the installation of the OPC server it should be offered for selection by the OPC client (for example visualization). The name of the OPC server is **CoDeSys.OPC.DA**.

The OPC server will be started automatically by the operating system as soon as a client establishes a connection. The OPC server will terminate automatically as soon as the clients have closed their connections to the server.

There will be no OPC server icon in the task bar. It will only appear in the **Windows Task Manager** as a process.

Executing the OPC Client on a PC not Running SoMachine

To be able to execute the OPC client on a PC where SoMachine is not installed, proceed as follows:

- Refer to the chapter *Installation of the CoDeSys OPC Server* in the *SoMachine Installation and Configuration Manager User Guide*, and execute the following actions:
 - Install the gateway on the PC where the OPC client is running.
 - Depending on the OPC client, you need to launch the *WinCoDeSysOPC.exe* file.
- Copy the file *OPCServer.ini* to the same directory where the *WinCoDeSysOPC.exe* is installed.

Appendix C

Script Language

What Is in This Chapter?

This chapter contains the following sections:

Section	Topic	Page
C.1	General Information	844
C.2	Schneider Electric Script Engine Examples	854
C.3	CoDeSys Script Engine Examples	869

Section C.1

General Information

What Is in This Section?

This section contains the following topics:

Topic	Page
Introduction	845
Executing Scripts	848
Best Practices	850
Reading .NET API Documentations	851
Entry Points	853

Introduction

SoMachine Script Language

The SoMachine script language provides a powerful tool to automatize sequences. You can start single commands or complex command sequences directly from the SoMachine program environment or from the Windows command line. The SoMachine script language is a modular language based on IronPython 2.7. The IronPython interpreter is integrated into the SoMachine development environment. This implementation allows using the extensive framework libraries of Python. Among other things, they provide access to files in networks.

This chapter does not provide an example for each member of the Script Engine because this would go beyond the scope of this document. The examples provided here have been selected due to a certain logic that runs like a common thread through the API (Application Programming Interface). If you follow this thread, you can find further members in the document *Automation Platform SDK* that is provided as online help on the CoDeSys webpage. You will then be able to use these members in the same manner as described for the examples in this chapter.

Programming Environments for Python

The following free IDE (Integrated Development Environments) are available:

IDE	Features
Notepad++	● Syntax highlighting. ● Universal editor.
IDLE (Integrated DeveLopment Environment)	● Syntax highlighting.
Visual Studio plugin	● Syntax highlighting. ● IntelliSense for IronPython language constructs (and .NET). ● Integrated debugging function. ● Easy to use. ● Requires Visual Studio.
PyCharm	● Syntax highlighting. ● IntelliSense ● Integrated debugging function. ● Easy to use. ● Standalone software product. ● Free Community Edition version 3.0 and later supporting numerous features such as: ○ For pure Python coding and learning. ○ Intelligent editor, with code completion, on-the-fly error highlighting, auto-fixes, etc. ○ Automated code refactorings and rich navigation capabilities. ○ Integrated debugger and unit testing support. ○ Native VCS (Version Control System) integrations. ○ Customizable user interface and key-bindings, with VIM (Vi IMproved text editor) emulation available.

Notes on Compatibility

Python is a dynamic language used for a wide variety of purposes with an emphasis on clean and expressive code. It allows the maximum flexibility for the developer, while maintaining readability of code. IronPython brings Python to .NET and allows native access to the .NET framework and classes. The implementation of the IronPython interpreter is based on Python version 2.7. There are many free tutorials and online help systems available on the Internet.

NOTE: Version incompatibility to Python V3.x. Keep in mind that the language Python uses may be upgraded, and if so, may delete old, deprecated language constructs. Therefore, write your scripts in a forward-compatible way. This involves, for example, using the `from __future__ import print_function` statement.

For more information, refer directly 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. Schneider Electric and CoDeSys have agreed on accepting the *Style Guide for Python Code*. Any new scripts should also comply with this standard.

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>.
- Wikipedia article *Python (programming language)* at [http://en.wikipedia.org/wiki/Python_\(programming_language\)](http://en.wikipedia.org/wiki/Python_(programming_language)).
- Official CoDeSys forum providing examples and helpful information at <http://forum.codesys.com/>.
- IronPython interpreter at <http://ironpython.codeplex.com/>.
- PyTools (Visual Studio plugin) at <http://pytools.codeplex.com/>.
- *IronPython Cookbook* at <http://www.ironpython.info/index.php/Contents>.
- Free Galileo Openbook (only available in German) at <http://openbook.galileocomputing.de/python/>.

Executing Scripts

Overview

You can execute script files (*filename.py*), containing a sequence of script commands, from the SoMachine user interface.

For further information on running scripts from the SoMachine user interface, refer to the chapter *Script-Related Commands* (see *SoMachine*, *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 SoMachine is installed
cd "<Replace this with the path to the Central.exe, for example,
C:\Program Files (x86)\Schneider Electric\SoMachine Software\>"
REM Run Central.exe with no graphical user interface and the full path
to the script
Central.exe --noui --runscript="<Replace this with the full file path
where the script is stored, for example, D:\MyScripts\TestScript.py>"
pause
```

C# Console Application

Running the script in a C# application allows you to edit the script dynamically before the script is executed by the engine. In addition, some previous steps can be performed in the C# application as well

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Diagnostics;
namespace ExecuteScriptExample
{
    class Program
    {
        static void Main(string[] args)
        {
            try
            {
                ProcessStartInfo psi = new ProcessStartInfo();

                // Specify the name and the arguments you want to pass
                psi.FileName = @"<Replace this with the path to the
Central.exe, for example,
C:\Program Files (x86)\Schneider Electric\SoMachine Software\>Central.e
xe>";

                psi.Arguments = "--noui --enablescripttracing --
AdvancedPythonFunctions --runscript=\
"<Replace this with the full file path where the script is stored,
for example, D:\MyScripts\TestScript.py>"";
                // Create new process and set the starting information
                Process p = new Process();
                p.StartInfo = psi;
                // Set this so that you can tell when the process has
completed
                p.EnableRaisingEvents = true;
                p.Start();
                // Wait until the process has completed
                while (!p.HasExited)
                {
                    System.Threading.Thread.Sleep(1000);
                }
            }
            catch (Exception ex)
            {
                Console.Write(ex.Message);
            }
            Console.ReadKey();
        }
    }
}

```

Best Practices

Best Practices

Consider the following practices that can help you to avoid errors in your program code:

- Function blocks are not created using curly brackets ({}) but by indentation of blocks of code.

Examples:

C Programming language	Python programming language
<pre>int factorial(int x) { if (x > 1) return x* factorial(x - 1); else return 1; }</pre>	<pre>def factorial(x): if x > 1: return x * factorial(x - 1) else: return 1</pre>

- Be careful when using copy and paste commands.
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.
To help to avoid this, configure your editor in such a way that it automatically replaces tabs by spaces.
- Consider case sensitivity (even in the command line).
- Be careful when entering path names.
- Do not forget to enter the closing quotation marks at the end of a command line (- - runscript="").
- Be careful with control characters that are frequently used in path names. A control character is preceded by a backslash (\) character. In order to deactivate this effect, insert an r in front of the quotation mark.

Example:

```
project_path = r"D:\PythonProjects\SetParameter.project"
```

- Make sure that loop declarations and conditions end with a colon.

Example:

```
if len(messages) == 0:
    print("--- Build successful ---")
```

Reading .NET API Documentations

Reading .NET API Documentations for Python Programmers

The current preliminary version of the script interface documentation is auto-generated from the underlying .NET / C# sources. Thus, it contains some idioms which are not common to Python programmers.

The list gives some hints on how to translate them to the Pythonic way of thinking:

- Interfaces in .NET are a contract about which members (methods, properties) have to be provided by the classes implementing that interface. In IronPython, one can implement one or several .NET interfaces by deriving them as one does with base classes. When a member declared by the interface is missing in the declaration, an exception will be thrown at runtime. (The *DeviceImportFromSvn.py* example shows a class implementing the *ImportReporter* interface.)
- In .NET, all parameters, properties, and function return values are statically typed. The allowed type is annotated in front of the parameter name. For functions, the type of the return value is in front of the function name. Instances of subclasses are allowed when a parent class (or interface) is mentioned. `void` denotes a function without a return value.
- Methods may be overloaded, a class can have several methods with the same name, but they differ in the number and/or types of parameters. IronPython will automatically call the matching variant.
- The type `INT` can contain an integral number between -2,147,483,648... 2,147,483,647, `BOOL` is equal to the Python type `BOOL` (`TRUE` and `FALSE`), the type `STRING` is equal to the Python type `str` and `unicode` (which are equal in IronPython). `IDictionary<Object, Object>` denotes a normal Python dictionary. IronPython automatically converts between the Python and .NET types.
- If a type `T` derives from `IBaseObject<T>`, that type can be extended with more members by other plugins. The actual usages of that type `T` in parameters or return values will be marked with `IExtendedObject<T>`.
- The interface `IEnumerable<T>` describes any sequence (lists, arrays, generators) which yield only objects of type `T` (or subclasses). When the sequence yields an incompatible object, an exception will be thrown at runtime.
- The interface `ICollection<T>` describes a list which contains only objects of type `T` (or subclasses). When trying to add an incompatible object, an exception will be thrown.
- The syntax `params T [] name` is equal to the Python syntax `*name` for variable argument lists, but restricts the parameters to type `T` (or subclasses).

- Enumerations (ENUM) do not exist as a language construct in Python. They are used to define a fixed amount of constant values, for example, the days of a week. Enumeration values defined in .NET can be accessed in IronPython via `Name.Member` syntax (similar to static class members), for example, `OnlineChangeOption.Try`. There are several patterns of emulating enums in Python, for example, <http://pypi.python.org/pypi/enum/> or <http://www.ironpython.info/index.php/Enumerations>.
- Properties marked with `{ get; set; }` are read-write, properties only marked with `{ get; }` are read-only. They are similar to the `@property` decorator in Python.

For scripts the following entry points are available:

- `system`: Basic functionality for integration in the SoMachine system. This object provides all functions described under *ISystem Interface*, such as the exit of SoMachine, the access to the message window or the query if the `--noUI` mode is running by use of the `ui-present` command.
- `projects`: Basic functionality for project management. This object provides all functions described under *IScriptProjects Interface*, such as loading of projects and project archives. Furthermore, it is the entry point to the individual projects.
- `online`: Basic functionality for online connection to device. By use of the `create_online_application` method, the online object of an application object can be created. This online object allows login to controllers, starting applications and retrieving variable values.

Entry Points

Detailed Information About Entry Points

Driver	Name (Type)	Description
System	system (ISystem)	basic functionality for integration in the SoMachine system
	Severity	ENUM for news priority
	Guid	data type for G lobally u nique i dentifier
	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

Section C.2

Schneider Electric Script Engine Examples

What Is in This Section?

This section contains the following topics:

Topic	Page
Device Parameters	855
Compiler Version	857
Visualization Profile	858
Update Libraries	859
Clean and Build Application	860
Communication Settings	861
Reset Diagnostic Messages	862
Reboot the Controller	862
Convert Device	863
Comparing Projects	866
Advanced Library Management Functions	867
Accessing POUs	868

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

```
# We enable the new python 3 print syntax
from __future__ import print_function
# The path to the project
project_path = r"D:\PythonProjects\SetParameter.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
# To set a parameter you need a device, a parameter and a new value
that should be assigned to the parameter
# At first search for the SERCOSIII node...
sercosNode = proj.find('SERCOSIII', True)[0]
# ... and add a device named Robot_XK29 to the node, assuming that no
other device is below the SERCOSIII node, otherwise the index must be c
hanged
sercosNode.insert("Robot_XK29", 0, DeviceID(4096, "1003
0082", "1.36.2.2"), 'LXM62DxS')
# Now get the children of the SERCOS node, assuming that Robot_XK29 is
the only one, otherwise the index must be changed
Robot_XK29 = sercosNode.get_children(True)[0]
# Call the get_all_parameters() function, to get a complete list of all
parameters of that device object.
# A parameter can contain subparameter, which can be checked with
.HasSubElements property
parameter_list = treeobj.get_all_parameters()
# prints all parameters
```

```
for parameter in parameter_list:
    print("ID: " + parameter.Identifier + " Name: " + parameter.Visible
Name + " Value: " + parameter.Value + " ParameterSet: " +
str((parameter.GetAssociatedConnector).ConnectorId))
# Get the WorkingMode parameter:
# ID: 191 Name: WorkingMode Value: 1 ParameterSet: 1
working_mode = Robot_XK29.get_parameter(191, 1)
# Finally set the WorkingMode parameter to 2 = Deactivated
Robot_XK29.set_parameter(working_mode, "2")
```

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);
# Set the project as primary project
proj = projects.primary
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 = "3.5.0.20"
print("New compiler version: " + compiler_settings.active_compiler_version)
# Save project
projects.primary.save()
```

Visualization Profile

Overview

With the visualization profile extension, you can display the visualization profiles and you can set the active visualization profile of the 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\GetVisualizationProfile.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
# Print the active profile
print("Current visual profile: " + visualization_settings.active_profile_name)
# Get all available visualization profiles
profile_names = visualization_settings.get_all_visual_profile_names()
# Print the profiles
for visual_profile in profile_names:
    print (" - " + visual_profile)
# Set the profile to V1.35.12.0
visualization_settings.active_profile_name = "V1.35.20.0"
# Get the active profile
profile_name = visualization_settings.active_profile_name
# Print the active profile
print("New visual profile: " + profile_name)
# Save project
projects.primary.save()
```

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 of SoMachine.

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
# Search for die 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);
# Set the project as primary project
proj = projects.primary
# Fetch the active application.
app = proj.active_application
# Clean application
new_project.clean_application(app)
# Compile application and store compiler messages in a list
messages = new_project.compile_application(app)
# If messages == None the build was successful
if len(messages) == 0:
    print("--- Build successful ---")
# Otherwise print results
else:
    for i in messages:
        # If serverity == 'Script Engine Exception' the plugin caused
        an exception.
        # The text describes the exeption details.
        print(i.Serverity, i.Text)
```


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', 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_communica-
tion_address())
    system.ui.info("Test complete")
main()
```

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 *(see page 860)*).

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]
# Get all testelements from testseries "TS_Crank"
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(project)
    #find the controller named 'LMC' which shall be rebooted
    controller = project.find('LMC', True)[0]
    #reboot the controller
    controller.reboot_plc()
    system.ui.info("Test 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
defmain():
    # Set the project as primary project
    proj = projects.primary
    controller = proj.find('LMC', True)[0]
    drive = proj.find('Drive', True)[0]
    # test if controller can be converted using DeviceID
    x01c = DeviceID(4096, "1003 009D", "1.36.2.6")
    if controller.can_convert(x01c):
        system.ui.info("Conversion to LMCx01C possible")
    # test if drive can be converted using Parameters and module id
    if drive.can_convert(4096, "1003 0082", "1.36.2.6", "LXM52"):
        system.ui.info("Conversion to LXM52 possible")
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():
    if not projects.primary:
        system.ui.error("No active project. Please open the project PythonTestProject.projectarchive")
        return
    # Set the project as primary project
    proj = projects.primary
    controller = proj.find('LMC', True)[0]
    alternativeControllers = controller.get_alternative_devices()
    print("ALTERNATIVE DEVICES FOR LMC")
    for id in alternativeControllers:

```

```

        print(deviceid_to_string(id))
    drive = proj.find('drive', True)[0]
    alternativeDrives = drive.get_alternative_devices()
    print("ALTERNATIVE DEVICES FOR DRIVE")
    for id in alternativeDrives:
        print(deviceid_to_string(id))
    system.ui.info("Test complete. Please check the script output
window")
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', True)[0]
    drive = proj.find('Drive', True)[0]
    # converting the controller
    controller.convert(4096, "1003 009D", "1.36.2.6")
    # converting the drive
    drive.convert(DeviceID(4096, "1003 0082", "1.50.0.4"), "LXM52")
main()

```

Comparing Projects

Overview

There are several use cases where it is useful to have a script automatically comparing the contents of 2 projects. The Python project comparison function allows you to compare 2 projects. As a result it provides the information if the projects are different, as well as a detailed XML tree that reflects the project tree and shows the differences for each object.

Script Engine Example

```
from __future__ import print_function
def main():
    proj = projects.primary
    # compare the Primary Project to another Project on disk
    diff = proj.compare_to("CompTest_Right.project")
    write_diff(diff, "Diff1.xml")
    # compare, but ignore whitespaces, comments and properties
    diff = proj.compare_to("CompTest_Right.project", True, True, True)
    write_diff(diff, "Diff2.xml")
def write_diff(differences, filename):
    if differences.DifferenceFound:
        f = open(filename, 'wb')
        f.writelines(differences.ResultTree)
main()
```

Advanced Library Management Functions

Overview

SoMachine Logic Builder offers advanced functions for managing libraries, the so-called forward compatible libraries (*see SoMachine, 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

```
p = projects.primary
app = p.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 POU's

Overview

The following example shows how to print and manipulate the code of a POU. It is only available for textual programming languages.

Script Engine Example

```
if not projects.primary:
    system.ui.error('No primary project set')
p = projects.primary
pou = p.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)
```

Section C.3

CoDeSys Script Engine Examples

Overview

This chapter provides examples of frequently used members of the CoDeSysScript Engine. For a complete description of the members of each namespace, refer to the CoDeSys API description.

What Is in This Section?

This section contains the following topics:

Topic	Page
Project	870
Online Application	875
Objects	878
Devices	879
System / User Interface (UI)	881
Reading Values	883
Reading Values From Recipe and Send an Email	884
Determine Device Tree of the Open Project	886
Script Example 4: Import a Device in PLCOpenXML From Subversion	887

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 `IPProject` instance (refer to the specification in the document *Automation Platform SDK*) which can be used for further steps.

```
# Creates a new project
proj = projects.create("C:\PythonProjects\Example.project", True)
```

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.

```
# Load the project
proj = projects.open(project_path)
```

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.

```
# Save archive
projects.primary.save_archive("D:\Archive\Example.archive")
```

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.

```
device = proj.find('DRV_Lexium62', True)[0]
```

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.

```
system.trace
import os
project_path = r"D:\MyProjects\Example.project"
import_path = r"D:\MyProjects\ImportFiles"
# Close project if opened
if projects.primary:
    projects.primary.close()
proj = projects.open(project_path);
# Set the new project to primary
proj=projects.primary
files = os.listdir(import_path)
# 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 progress: ", name)
    def skipped(self, list):
        for obj in list:
            print("Skipped: ", obj.get_name())
```

```
def importFilter(name, guid,type,path):
    return True;
# create the importer instance.
handler = Handler()
for file in files:
    file_path = import_path + "\\\" + file
    print(file)
    proj.import_native(file_path, importFilter, handler)
proj.save();
```

PLCOpenXML Import

This method imports the contents of the specified PLCopenXML file into the top level of the project.

```
# CoDeSys XML import/export functionality.
from __future__ import print_function
import sys, io
# Set target Project to primary
proj=projects.primary
# 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()
filename = r"D:\ExportObjects\Drv_Master.xml"
# Search for the SERCOSIII node, where the device should be added.
device = proj.find('SERCOSIII', True)[0]
# Import the data into the project.
device.import_xml(reporter, filename)
```

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.

```
# Tests CoDeSys native import/export functionality.
from __future__ import print_function
project_path = r"D:\MyProjects\Example.project"
# Clean up any open Project:
if projects.primary:
    projects.primary.close()
proj = projects.open(project_path);
proj=projects.primary
import sys, io
# Collect all POU nodes in that list.
projectObjects = []
# Collect all the leaf nodes.
for node in proj.get_children(True):
    projectObjects.append(node)
# Print everything just to know what is going on.
for i in projectObjects:
    print("Found: ", i.type, i.guid, i.get_name())
# Export the files.
for candidate in projectObjects:
    # 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 = "D:\ExportFiles\\%s__%s.export" % (candidate.get_name(),
candidate.guid)
    # print some user information
    print("Exporting ", len(objects), " objects to: ", filename)
    # and actually export the project.
    proj.export_native(objects, filename)
```

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.

```
# CoDeSys XML import/export functionality.
from __future__ import print_function
import sys, io
# Set target Project to primary
proj=projects.primary
# define the printing function
def printtree(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):
        printtree(child, depth+1)
for obj in projects.primary.get_children():
    printtree(obj)
# 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
reporter = Reporter()
# Finds the DRV_Master object in the project
device = proj.find('DRV_Master', True)
filename = r"D:\ExportObjects\Drv_Master.xml"
# Exports the object to the hard drive
proj.export_xml(reporter, device, filename, True, True)
```

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.

```
app = online.create_online_application();
```

Prepared Values

This example shows how to write prepared values.

```
# Tests for the Online functionality - listing of prepared values.
import utils
try:
    print "Trying to create online application."
    app = online.create_online_application();
except Exception as e:
    print "Currently offline, executing startup..."
    execfile(utils.makepath("OnlineTestStartup.py"))
    print "Retrying to create online application."
    app = online.create_online_application();
print "app:", app.application_state, "op:",
app.operation_state.ToString("f")
print "Unpreparing values:"
for expression in app.get_prepared_expressions():
    app.set_prepared_value(expression, '')
    print "%s: '%s' '%s'" % (expression, app.read_value(expression),
app.get_prepared_value(expression))
print "Unforcing values:"
for expression in app.get_forced_expressions():
    app.set_unforce_value(expression)
    print "%s: '%s' '%s'" % (expression, app.read_value(expression),
app.get_prepared_value(expression))
app.force_prepared_values()
assert len(app.get_prepared_expressions()) == 0, "still some prepared
values remain..."
```

```
assert len(app.get_forced_expressions()) == 0, "still some prepared values remain..."
print "now preparing a value and forcing it:"
app.set_prepared_value("POU.testoutput", "4711");
app.force_prepared_values()
print "now preparing a value and writing it:"
app.set_prepared_value("POU.testint", "INT#1147");
app.write_prepared_values();
print "The prepared values are now written."
```

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.

```
project_path = r"D:\MyProjects\Example.project"
# Clean up any open Project:
if projects.primary:
    projects.primary.close()
proj = projects.open(PROJECT);
proj = projects.primary
# 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)
```

Logout Application

This method logs out the application. If the application is not logged in, nothing happens.

```
# Logout.onlineapp.logout()
```

Start Application

This method starts the application.


```
project_path = r"D:\MyProjects\Example.project"
# Clean up any open Project:
if projects.primary:
    projects.primary.close()
proj = projects.open(PROJECT);
proj = projects.primary
# 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)
```

Stop Application

This method stops the application.

```
# Stop the application
onlineapp.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.

```
# Finds the DRV_Master object in the project  
device = proj.find('DRV_Master', True)
```

Remove

This method removes the object.

```
# Finds the DRV_Master object in the project  
device = proj.find('DRV_Master', True)  
# Removes the DRV_Master object from the project  
device.remove()
```

Rename

This method renames the object to the new name.

```
# Finds the DRV_Master object in the project  
device = proj.find('DRV_Master', True)  
# Removes the DRV_Master object from the project  
device.rename('DRV_Master_2')
```

Import/Export

Refer to the description of imports/exports for projects ([see page 872](#)). 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
- a string specifying the module ID

```
device.insert("RobotA", DeviceID(4096, "1003 0082", "1.36.1.1"),
'LXM62DxS')
```

Disable

This method marks this device as disabled during download.

```
# Finds the DRV_Master object in the project
device = proj.find('DRV_Master', True)[0]
device.disable()
```

Enable

This method marks this device as enabled during download.

```
# Finds the DRV_Master object in the project
device = proj.find('DRV_Master', True)[0]
device.enable()
```

Get Address

This method gets the address of the device. It returns a string.

```
# Finds the DRV_Master object in the project
device = proj.find('DRV_Master', True)[0]
device.get_address()
```

Get Device Identification

This method gets the device identification.

```
# Finds the DRV_Master object in the project
device = proj.find('DRV_Master', True)[0]
deviceid = device.get_device_identification()
print("{0} - {1} {2}".format(" " * depth, name, deviceid))
```

Get Gateway

This method returns the GUID of the gateway.

```
# Finds the DRV_Master object in the project
device = proj.find('DRV_Master', True)[0]
gateway = 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
- a string specifying the module ID

```
device.insert("RobotA", 1, DeviceID(4096, "1003 0082", "1.36.1.1"),
'LXM62DxS')
```

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.

```
device.set_gateway_and_address(GUID gateway, string address)
```

Set Simulation Mode

This method sets the simulation mode. If set to True, simulation is enabled.

```
# Finds the DRV_Master object in the project
device = proj.find('DRV_Master', True)[0]
device.set_simulation_mode (True)
```

Update

This method updates the specified device.

```
# Finds the DRV_Master object in the project
device = proj.find('DRV_Master', True)[0]
device.update(DeviceID(4096, "1003 0082", "1.36.1.1"), 'LXM62DxS')
```

System / User Interface (UI)

Browse Directory Dialog Box

Opens a dialog box for browsing a directory. In --noUI mode, you can simply enter a path here.

```
system.ui.browse_directory_dialog("Browse Directory Dialog",
r"D:\Python", Environment.SpecialFolder.Desktop, True)
```

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.

```
list = ["RobotA", "RobotB", "RobotC"]
system.ui.choose("Choose", list, True)
```

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.

```
system.ui.info("Info")
```

Open File Dialog Box

This method displays an **Open File** dialog box. In --noUI mode, you can simply enter a path here.

```
system.ui.open_file_dialog("Select a file")
```

Query String

This method queries the input or edit of a text string.

```
system.ui.query_string("Please enter a string")
```

It returns a string with the entered text.

Save File Dialog

This method displays a **Save File** dialog box. In --noUI mode, you can simply enter a path here.

```
system.ui.save_file_dialog("Python Script: Save File")
```

Warning

This method indicates a warning message. It inhibits any further actions until the message has been acknowledged.

```
system.ui.warning("Warning")
```

Reading Values

Overview

You can start the following example from the SoMachine user interface or from the command line.

To start it from the command line, change to the subdirectory **Common** of the SoMachine installation path (*<Replace this with the path to the Central.exe, for example, C:\Program Files (x86)\Schneider Electric\SoMachine Software\>*) and enter the command `start /wait Central.exe --`

`runscript=<Replace this with the full file path where the script is stored, for example, D:\MyScripts\ReadVariable.py>`

The script opens an application in SoMachine and logs in to the device. If the controller is not in mode RUN, it will be set to RUN. 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*

```
# Close all projects
while len(projects.all) > 0:
    projects.all[0].close()
# opens project
proj = projects.open("D:\\data\\projects\\Ampel.project")
# set "Ampel.project" to active application
app = proj.active_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("PLC_PRG.iVar1")
# display value in message view or command line
print value
# log out from device and close "Ampel.project"
onlineapp.logout()
proj.close()
```

Reading Values From Recipe and Send an Email

Overview

You can start the following example from the SoMachine user interface or from the command line.

To start it from the command line, change to the subdirectory **Common** of the SoMachine installation path (*<Replace this with the path to the Central.exe, for example, C:\Program Files (x86)\Schneider Electric\SoMachine Software\>*) and enter the command `start /wait Central.exe --`

`runscript=<Replace this with the full file path`

`where the script is stored, for example, D:\MyScripts\ScriptEmail.py>".`

The script opens an application in SoMachine and logs in to the device. If the controller is not in mode RUN, it will be set to RUN. 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*

```
# Close current project if necessary and open "ScriptTest.project"
if not projects.primary == None:
    projects.primary.close()
project = projects.open("D:\\Data\\projects\\scriptTest.project")
# retrieve active application
application = project.active_application
# create online application
online_application = online.create_online_application(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)
# open recipe file to read values.
recipe_input_file = open("D:\\Data\\projects\\RecipeInput.txt", "r")
watch_expressions = []
for watch_expression in recipe_input_file:
    watch_expressions.append(watch_expression.strip())
print watch_expressions
# read values from the controllerd
watch_values = online_application.read_values(watch_expressions)
print watch_values
# open output file to write values
recipe_output_file = open("D:\\Data\\projects\\RecipeOutput.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_input_file.close()
recipe_output_file.close()
# send Email
# import respective libraries
import smtplib
from email.mime.text import MIMEText
#open output file
recipe_output_file = open("D:\\Data\\projects\\RecipeOutput.txt", "r")
mail = MIMEText(recipe_output_file.read())
recipe_output_file.close()
#email address sender and recipient
fromm = "info@3s-software.com"
to = "info@3s-software.com"
# set sender and recipient
mail["Subject"] = "Attention value has changed"
mail["From"] = fromm
mail["To"] = to
# send email
smtp = smtplib.SMTP("name of smtp server")
smtp.sendmail(fromm, [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 SoMachine user interface or from the command line.

To start it from the command line, change to the subdirectory **Common** of the SoMachine installation path (*<Replace this with the path to the Central.exe, for example, C:\Program Files (x86)\Schneider Electric\SoMachine Software\>*) and enter the command `start /wait Central.exe -- runscript=<Replace this with the full file path where the script is stored, for example, D:\MyScripts\DevicePrintTree.py>`.

Script Example *DevicePrintTree.py*

```
# We enable the new python 3 print syntax
from __future__ import print_function
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.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!", file=sys.stderr)
    sys.exit()
# And the actual output
print("--- The current tree of: ---")
printtree(projects.primary)
print("--- Script finished. ---")
```

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 SoMachine user interface or from the command line.

To start it from the command line, change to the subdirectory **Common** of the SoMachine installation path (*<Replace this with the path to the Central.exe, for example, C:\Program Files (x86)\Schneider Electric\SoMachine Software\>*) and enter the command `start wait Central.exe runscript="<Replace this with the full file path where the script is stored, for example, D:\MyScripts\DeviceImportFromSvn.py>"`.

Script Example *DeviceImportFromSvn.py*

```
# Imports a Device in PLCOpenXML from Subversion via command line svn
client.
# We enable the new python 3 print syntax
from __future__ import print_function
import sys, os
# some variable definitions:
SVNEXE = r"C:\Program Files\Subversion\bin\svn.exe"
XMLURL = "file:///D:/testrepo/testfolder/TestExport.xml"
PROJECT = r"D:\test.project"
# 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('"' + SVNEXE + '" cat ' + XMLURL, 'r') as pipe:
    xmldata = pipe.read()
# create a new project:
proj = projects.create(PROJECT)
# 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.
proj.import_xml(reporter, xmldata)
# and finally save. :-)
proj.save()
print("--- Script finished. ---")
```

Appendix D

User Management for Soft PLC

Overview

This chapter describes the **Users and Groups** and the **Access Rights** views of the device editor. These views are only available if the SoMachine project contains **Soft PLC** controllers.

What Is in This Chapter?

This chapter contains the following topics:

Topic	Page
General Information on User Management for Soft PLC	890
Users and Groups	892
Access Rights	895

General Information on User Management for Soft PLC

Overview

The user management function described in this chapter allows you to define user accounts and to configure the access rights (permissions) for **Soft PLC** controllers.

The rights to access project objects via specified actions are assigned only to user groups, not to a single user account. So each user must be member of a group.

User Management for Soft PLC

Before setting up users and user groups for **Soft PLC** controllers, consider the following:

- By default, a group **Everyone** exists. Each defined user or other groups is automatically member of this group. Thus, each user account at least is automatically provided with default settings. Group **Everyone** cannot be deleted, only renamed. Members cannot be removed from this group. By default, **Everyone** does not have the permission to modify the current users, groups, and permission configuration.
- By default, a group **Owner** also exists, containing one user **Owner**. In a new project, initially only the **Owner** has the permission to modify the current users, groups, and permission configuration. Thus, only **Owner** can assign this right to another group. Initially, the **Owner** can log in with username **Owner** and empty password. Users can be added to or removed from group **Owner**, but at least 1 user must remain. This group - such as **Everyone** - cannot be deleted. It is granted all access rights. Thus, it is not possible to make a project unusable by denying the respective rights to all groups. You can rename both group and user **Owner**.
- When starting the programming system or starting a project, primarily no user is logged on the project. But then the user can optionally log on via a defined user account with user name and password in order to have a special set of access rights.
- Consider that each project has its own user management. So, for example to get a special set of access rights for a library included in a project, the user must separately log on to this library. Also users and groups, set up in different projects, are not identical even if they have identical names.

NOTE: Only the user **Owner** of group **Owner** is allowed to modify the currently configured permissions, groups, and users. Thus, only **Owner** can assign this permission to another group.

NOTE: The user passwords are stored irreversibly. If you do not remember the password, the respective user account becomes unusable. If you do not remember the password of the **Owner** group, the entire project can become unusable.

Access Right Management for Soft PLC

User management in a project is only useful in combination with the access right (=permissions) management.

Consider the following:

- In a new project, basically all rights are not yet defined explicitly but set to a default value. This default value usually is granted with exception of the right to modify the current users, groups, and permission configuration. By default, this is only granted for the **Owner** group.
- When the project is created, a member of the group with the right to modify the permissions can define rights. Each particular right can be granted or denied or set back to default.
 - Perform the access right management of a project in the **Project → User Management → Permissions... → Permissions** dialog box.
 - Perform the access right management for objects in the **View → Properties... → Properties - Project Information** dialog box, selecting the **Access control** tab.
- Access rights on objects are inherited. If an object has a father object (for example, if an action is assigned to a program object, that is inserted in the structure tree below the program, then the program is the father of the action object), the current rights of the father will automatically become the default settings of the child. Father-child relations of objects concerning the access rights usually correspond with the relations shown in the **Devices tree**, **Applications tree**, and **Tools tree** and are indicated in the **Permissions** dialog box by the syntax <father object>.<child object>. Example: Action ACT is assigned to POU object PLC_PRG. So in the **Applications tree** ACT is shown in the tree structure indented below PLC_PRG. In the **Permissions** dialog box, ACT is represented by PLC_PRG . ACT indicating that PLC_PRG is the father of ACT. If the modify right is denied explicitly for PLC_PRG and a certain user group, the default value of the modify right for ACT will also be automatically denied.

Users and Groups

Overview

The **Users and Groups** view of the device editor is provided for devices supporting online user management. It allows setting up user accounts and user groups, which, in combination with the access right management, serves to control the access on objects on the controller in online mode.

If it is desired that certain functions of a controller can only be executed by authorized users, use the online user management feature. It allows you to set up user accounts to assign access rights for user groups and to force a user authentication at login.

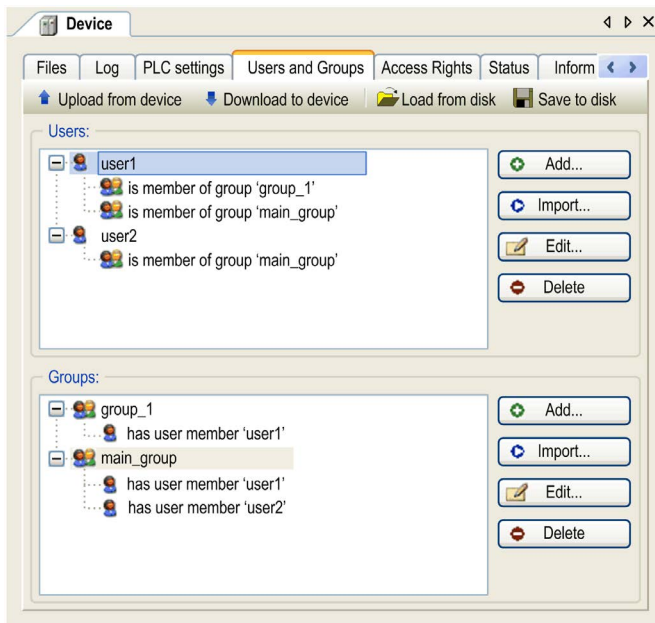
The device-specific user management can be pre-defined by the device description. This description also defines to which extent the definitions can be edited in the configuration dialog boxes

As in the project user management users have to be members of groups. Only user groups can obtain certain access rights (*see page 895*).

Using the Configuration Dialog

Basically, the handling of the user management dialogs is similar to that of the project user management. There is even the possibility to import user account definitions from the project user management.

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**.

Users Area

The following buttons are available for setting up user accounts:

Button	Description
Add	Opens the dialog box Add User . Enter a user Name and a Password . Repeat the password in the Confirm password field.
Import	Opens the dialog box Import Users . It shows all user names which are defined in the project user management. Select 1 or several entries and click OK to confirm. The dialog box Enter password opens. Enter the corresponding password as it is defined in the project user management. You can then import the user account to the device-specific user management. However, these passwords are not imported. NOTE: Each imported user account will have an empty password definition.
Edit	Modifies the selected user account concerning user name and password. The Edit User <user name> dialog box corresponds to the Add User dialog box (see above).
Delete	Deletes the selected user account.

Groups Area

The following buttons are available for setting up user groups:

Button	Description
Add	Opens the dialog box Add Group . Enter a group Name and select from the defined users those who should be members of this group.
Import	Opens the dialog box Import Groups . It shows the groups which are defined in the project user management. Select 1 or several entries and click OK to integrate them in the group list of the device-specific user management.
Edit	Modifies the selected group concerning group name and associated users. The Edit Group <group name> dialog box corresponds to the Add Group dialog box (see above).
Delete	Deletes the currently selected group.

Applying and Storing the Current Configuration

The buttons are available in the top bar of the dialog box:

Button	Description
Download to device	Downloads the current user management configuration to the device. It will become effective only after it has been downloaded.
Upload from device	Uploads the configuration currently applied on the device into the configuration dialog box.
Save to disk / Load from disk	The current configuration can be stored in an XML file (*.DUM) and reloaded from this file. This is useful to set up the same user configuration on multiple systems. The standard dialog for browsing in the file system is provided for this purpose. The file filter is automatically set to *.DUM, which means device user management files.

Printing the User Management Configuration

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

Access Rights

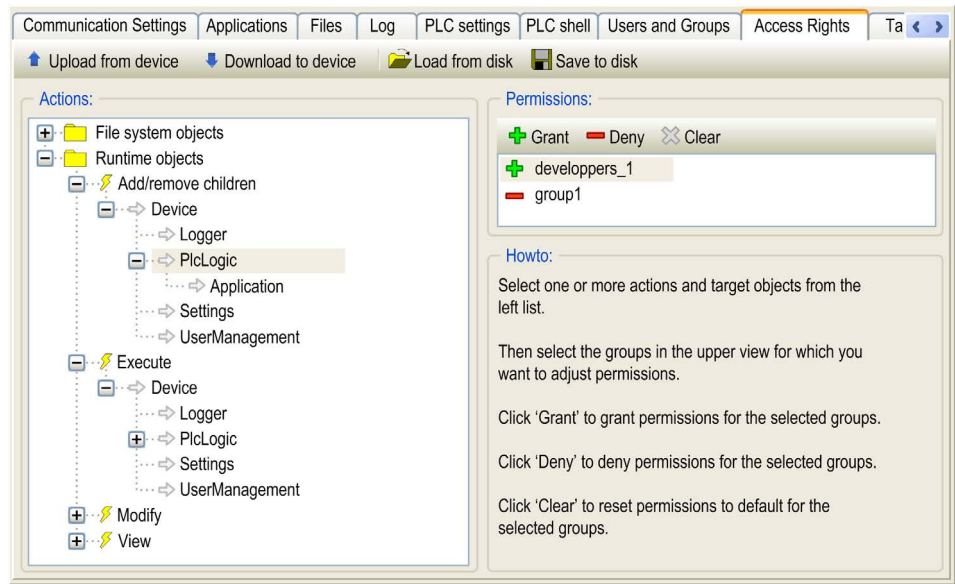
Overview

The **Access Rights** view of the device editor is part of the online user management feature (see page 892). It serves to grant or deny the currently defined user groups certain permissions, thus defining the access rights for users on files or objects (for example, an application) on the controller during runtime.

Consider that the access permissions can only be assigned to groups, not to particular users. For this reason, you have to define a user as member of a group. Do the configuration of users and groups in the **Users and Groups** view of the device editor (see page 892).

See the example in the following image: the permission to add and remove children to/from the **PlcLogic** object granted for user group **developpers_1**.

Access Rights view of the device editor



Defining Access Permissions

To define the permission for performing an action on 1 or multiple objects, proceed as follows:

Step	Action
1	Select the object entry or entries below the desired action type in the Actions area.
2	Select the desired group in the Permissions area.
3	Click the Grant or Deny button.

NOTE: When setting an access right on an object, consider the mapping table shown in the *Which Action in Detail is Concerned by a Certain Access Right on a Certain Object?* paragraph (see page 897).

See the instructions in the **Howto** area of the view.

Actions Area

The **Actions** area lists the actions which can be performed during runtime on files in the controller file system or runtime objects, for example, applications. The tree is structured in the following manner:

On the top level there are 2 object categories grouped in folders:

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

Indented below the object categories, there are nodes for the 4 types of actions. They can be performed on the particular objects.

- **Modify** (for example, downloading application)
- **View** (monitoring)
- **Add/remove children** (adding or removing of child objects to/from an existing object)
- **Execute** (for example, start/stop application, setting breakpoints)

Below each action type node, find the objects, or targets of the action. These are the controller file or runtime objects, for example, **Device**.

These object entries are displayed in a tree structure mapping the device tree or the file system structure.

NOTE: Assigning an access right definition to a father node in the objects tree usually means that the children nodes will inherit this definition. This is valid as long as they do not apply their own explicit definition. However, depending on the device, this can be handled differently. In any event, inheritances are not visualized here in the view.

Permissions Area

The **Permissions** area shows the defined user groups.

Before each group, one of the following icons indicates the currently assigned permission concerning the target which is selected in the **Actions** area:

Icon	Description
– (minus sign)	The actions selected in the Actions area are granted for the group.
+ (plus sign)	The actions selected in the Actions area are denied for the group.
X (cross sign)	There is no explicit access right definition for the actions selected in the Actions area.
no icon being displayed	Multiple actions are selected in the Actions area which do not have unique settings referring to the currently selected group.

After you have selected the desired objects below the desired action in the **Actions** area and you have selected the desired group in the **Permissions** area, you can use the following buttons:

Button	Description
Grant	Explicit granting access permission
Deny	Explicit denying access permission
Clear	The granted access right for the actions currently selected in the Actions area will be deleted that is set back to the default.

Applying and Storing the Current Configuration

The buttons are available in the top bar of the dialog box:

Button	Description
Download to device	Downloads the currently configured access right definitions to the device. They will become effective only after they have been downloaded.
Upload from device	Uploads the access rights currently applied on the device into the configuration dialog box.
Save to disk / Load from disk	The current configuration can be stored in an XML file (*.DAR) and reloaded from this file. This is useful to set up the same user configuration on multiple systems. The standard dialog for browsing in the file system is provided for this purpose. The file filter is automatically set to *.DAR, which means device access right files.

Printing the Access Rights Definition

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

Which Action in Detail Is Concerned by a Certain Access Right on a Certain Object?

Objects			Action	Rights			
				Add/ Remove Children	Execute	Modify	View
X = right must be set explicitly – = right is not relevant							

Objects		Action	Rights						
Device		Login	–	–	–	X			
	Logger	Read entries	–	–	–	X			
	PlcLogic	Application	Login	–	–	–	X		
			Create	–	–	X	–		
			Create child	X	–	X	–		
			Delete	–	–	X	–		
			Download / Online Change	–	–	X	–		
			Create bootproject	–	–	X	–		
			Read variable	–	–	–	X		
			Write variable	–	–	X	X		
			Force variable	–	–	X	X		
			Set + delete breakpoint	–	X	X	–		
			Set next statement	–	X	X	–		
			Read callstack	–	–	–	X		
			Single cycle	–	X	–	–		
			Set flow control	–	X	X	–		
			Read flow control	–	–	–	X		
			Run / Stop	–	X	–	–		
			Reset	–	–	X	–		
			Settings		Read settings	–	–	–	X
					Write settings	–	–	X	–
	UserManagement		Read configuration	–	–	–	X		
			Write configuration	–	–	X	–		

X = right must be set explicitly
– = right is not relevant

Appendix E

Controller Feature Sets for Migration

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
TWDLCCA10DRF	6	4	No	3	1	0	1	No
TWDLCAA16DRF	9	7	No	3	1	0	1+1	No
TWDLCCA16DRF	9	7	No	3	1	0	1+1	No
TWDLCAA24DRF	14	10	No	3	1	0	1+1	No
TWDLCCA24DRF	14	10	No	3	1	0	1+1	No
TWDLCAA40DRF	24	16	No	4	2	2	1+1	No
TWDLCCA40DRF	24	16	No	4	2	2	1+1	No
TWDLCAE40DRF	24	16	No	4	2	2	1+1	Yes
TWDLCCDE40DRF	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
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

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

Controller	Dig In	Dig Out	Ana In	MOD	FC	HSC	PWM	Serial	ETH	CART
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										

SoMachine Controllers

Controller	Dig In	Dig Out	Ana In	MOD	FC	HSC	PWM	Serial	ETH	CART
TM241C24R	14	10	0	TM2/ TM3	4	2	2	2	No	1
TM241C24T/U	14	10	0	TM2/ TM3	4	2	2	2	No	1
TM241C40R	24	16	0	TM2/ TM3	4	2	2	2	No	2
TM241C40T/U	24	16	0	TM2/ TM3	4	2	2	2	No	2
TM241CE24R	14	10	0	TM2/ TM3	4	2	2	2	Yes	1
TM241CE24T/U	14	10	0	TM2/ TM3	4	2	2	2	Yes	1
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										

Controller	Dig In	Dig Out	Ana In	MOD	FC	HSC	PWM	Serial	ETH	CART
TM241CE40R	24	16	0	TM2/ TM3	4	2	2	2	Yes	2
TM241CE40T/U	24	16	0	TM2/ TM3	4	2	2	2	Yes	2
TM241CEC24R	14	10	0	TM2/ TM3	4	2	2	2	Yes	1
TM241CEC24T/U	14	10	0	TM2/ TM3	4	2	2	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										



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.

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.

FDT

(*field device tool*) The specification describing the standardized data exchange between the devices and control system or engineering or asset management tools.

G

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 a SoMachine 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.

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.

R

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.)

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.



Symbols

- IronPython, *845*
- __DELETE
 - operator, *717*
- __ISVALIDREF
 - operator, *719*
- __NEW
 - operator, *720*
- __QUERYINTERFACE
 - operator, *723*
- __QUERYPOINTER
 - operator, *725*

0-9

- 3 GB switch on Windows 7 32-bit, *803*
- 32-bit operating system
 - memory limit, *803*

A

- ABS
 - IEC operator, *703*
- ACOS
 - IEC operator, *712*
- ADD
 - IEC operator, *641*
- adding controllers, *64*
- adding controllers by drag and drop, *60*
- adding devices and modules by drag and drop, *61*
- adding devices from device template by drag and drop, *62*
- adding devices from function template by drag and drop, *62*
- adding expansion devices by drag and drop, *61*
- addressing, *823*
- ADR
 - operator, *681*

- analog inputs
 - CANopen, *799*
- AND
 - IEC operator, *655*
- ANY_NUM_TO
 - IEC operator, *701*
- ANY_TO
 - IEC operator, *701*
- array, *633*
- ASIN
 - IEC operator, *711*
- ATAN
 - IEC operator, *713*
- automatic I/O mapping, *143*

B

- BIT, *606*
- BITADR
 - operator, *683*
- BOOL, *601*
- BOOL_TO
 - IEC operator, *687*
- boot application, *239*
- Boot Application, *233*
- build-time performance, *804*

C

CAL

- IEC operator, *684*

- CANopen analog inputs, *799*

- CANopen devices, *66*

CASE

- instruction, *370*

- Catalog view, *44*

command

- Convert Device, *73*

- Convert SoMachine Basic Project, *77*

- Convert Twido Project, *77*

- communication manager configuration, *66*

- communication settings, *103*

- configuration diagnostic, *69*

- constants, *541*

Content

- operator, *682*

CONTINUE

- instruction, *370*

- controller - HMI variable exchange, *510*

- Convert Device command, *73*

- convert SoMachine Basic project, *77*

- convert Twido project, *77*

COS

- IEC operator, *709*

D

- data types, *601*

- DATE, *603*

- DATE_AND_TIME, *603*

DATE_TO

- IEC operator, *696*

- device templates, *767, 768*

devices

- adding, *68*

DIV

- IEC operator, *647*

- download, *232, 239*

- boot application, *239*

- Boot Application, *233*

- DT, *603*

DT_TO

- IEC operator, *696*

- DTM, *39*

E

EQ

- IEC operator, *678*

EXIT

- instruction, *370*

EXP

- IEC operator, *707*

- expansion devices, *65*

- expansions, *65*

EXPT

- IEC operator, *714*

- external variables, *538*

F

FAQ

- login to controller not successful, *814*

FB_init

- IEC operator, *729*

- FdtConnections node, *39*

- FFB Finder, *436*

- field device configuration, *66*

- fieldbus health information, *68*

- fieldbusses supported by templates, *757*

FOR

- instruction, *370*

- Function and Function Block Finder, *436*

- function templates, *781*

G

GE

- IEC operator, *677*

- global variables, *537*

GNVL

- global network variables list, *403*

GT

- IEC operator, *674*

H

health information
 fieldbus, *68*
 HMI - controller variable exchange, *510*
 HMI controller
 unsuccessful multiple download, *806*

I

IEC objects
 fieldbus Diagnostic/O mapping, *68*
 IF
 instruction, *370*
 increasing build-time performance, *804*
 INI
 IEC operator, *729*
 input and output variables, *537*
 input variables, *536*
 installation
 third-party Sercos devices, *57*
 instructions
 ST editor, *370*
 INT_TO
 IEC operator, *691*
 integer, *601*

J

JMP
 instruction, *370*

L

large SoMachine projects, *803*
 LE
 IEC operator, *676*
 LIMIT
 IEC operator, *671*
 literals
 typed, *541*
 LN
 IEC operator, *705*
 local variables, *536*

LOG

 IEC operator, *706*
 login, *232, 239*
 login to controller not successful, *814*
 LREAL, *602*
 LREAL_TO
 IEC operator, *692*
 LT
 IEC operator, *675*
 LTIME, *605*

M

managing tags, *44*
 MAX
 IEC operator, *669*
 memory consumption of SoMachine, *804*
 memory limit, *803*
 menus, *801*
 MIN
 IEC operator, *670*
 MOD
 IEC operator, *650*
 Modbus IOScanner on a Serial Line, *805*
 Modbus Serial IOScanner
 error detected, *805*
 Modbus SL devices, *66*
 Modbus slave disconnected, *805*
 MOVE
 IEC operator, *652*
 MUL
 IEC operator, *643*
 multiple download not working on HMI controller, *806*
 MUX
 IEC operator, *672*

N

NE

- IEC operator, *679*

Network Device Identification

- accessing new controllers, *809*
- connecting via IP address and address information, *811*
- FAQs, *814*
- in Controller Selection view, *99*

NodeName, *105*

NOT

- IEC operator, *658*

NVL

- configuration example, *409*
- considerations, *399*
- controllers supporting NVL, *399*
- network variables list, *398*
- rules, *406*

NVL communication suspended, *806*

O

OPC client

- variables mapped to %I , *835*

OPC server, *830*

OPC Server 3, *829*

OPC server configuration, *836*

OR

- IEC operator, *656*

output variables, *536*

P

persistent variables, *540*

Process communication settings dialog box, *103*

programming environments for Python, *845*

publishing variables, *504*

publishing variables (HMI), *507*

Python, *845*

R

REAL, *602*

REAL_TO

- IEC operator, *692*

reducing memory consumption, *804*

refreshing variables, *510*

remanent variables, *539*

REPEAT

- instruction, *370*

retain variables, *539*

RETURN

- instruction, *370*

ROL

- IEC operator, *663*

ROR

- IEC operator, *665*

routing, *823*

run, *249*

S

Script Engine, *845*

searching within catalogs, *44*

SEL

- IEC operator, *668*

selecting variables, *506*

Sercos

- installation of third-party devices, *57*

SHL

- IEC operator, *660*

shortcuts, *801*

SHR

- IEC operator, *662*

SIN

- IEC operator, *708*

SIZEOF

- IEC operator, *653*

smart coding, *596*

SQRT

- IEC operator, *704*

ST editor

- instructions, *370*

startup performance of SoMachine, *800*

static variables, *538*

stop, *249*

STRING, *603*

STRING_TO
 IEC operator, *698*
SUB
 IEC operator, *645*
symbol configuration, *496*

T

tagging catalog items, *44*
TAN
 IEC operator, *710*
tasks
 adding, *229*
template libraries, *767*
templates, *754*
temporary variables, *538*
TIME, *603*
time data types, *603*
TIME_OF_DAY, *603*
 IEC operator, *694*
TIME_TO
 IEC operator, *694*
TO_BOOL
 IEC operator, *689*
TOD, *603*
TRUNC
 IEC operator, *700*
TRUNC_INT
 IEC operator, *701*
typed literals, *541*

U

UNION, *605*
updating devices, *71*
Use DTM Connection checkbox, *39*
user management, *128*
Users and Groups, *128*

V

variable definition, *500*
variable exchange
 communication speed, *510*
variable types, *500*

variables, *536*
 persistent, *213*
 publishing, *504*
 publishing (HMI), *507*
 remanent, *213*
visualizations, *772*

W

WHILE
 instruction, *370*
WSTRING, *606*

X

XOR
 IEC operator, *657*

