

BECKHOFF New Automation Technology

Manual | EN

TE1000

TwinCAT 3 | PLC

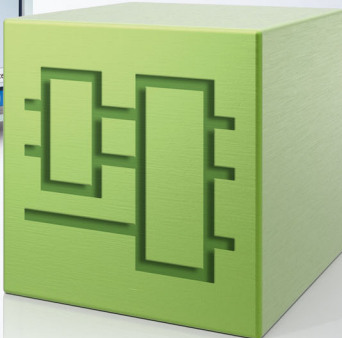
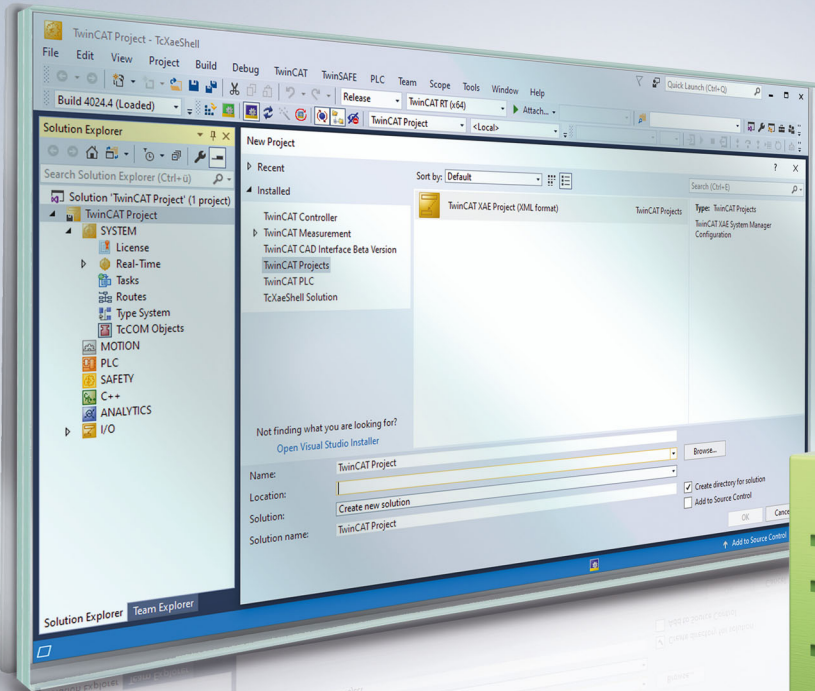


Table of contents

1 Foreword	19
1.1 Notes on the documentation	19
1.2 For your safety	19
1.3 Notes on information security.....	21
2 Quickstart	22
2.1 Differences compared with TwinCAT 2.....	23
2.2 Your first TwinCAT 3 PLC project	24
3 Tips and tricks	38
3.1 Project delivery.....	38
3.2 New properties and features	42
3.2.1 TwinCAT 3.1 Build 4024	42
3.2.2 TwinCAT 3.1 Build 4026	46
3.3 Further helpful properties and features	48
3.3.1 Hotkey.....	51
3.4 Renaming a project	52
4 Creating and configuring a PLC project	54
4.1 Creating a standard project.....	54
4.2 Adding objects.....	55
4.3 Changing the compiler version.....	56
4.4 Open a TwinCAT 3 PLC project.....	57
4.5 Open a TwinCAT 2 PLC project.....	57
4.6 Configuring a PLC project.....	60
4.7 Using global data types	61
5 Exporting and transferring a PLC project	62
5.1 Exporting and importing a PLC project	62
5.2 Transferring a PLC project	63
6 Localizing the PLC project	64
7 Programming a PLC project	66
7.1 Assigning identifiers	66
7.2 Declaring variables.....	66
7.2.1 AT-Declaration	69
7.2.2 Using the Declaration Editor	70
7.2.3 Using the Auto Declare dialog.....	71
7.2.4 Declaring an array.....	71
7.2.5 Declaring global variables	72
7.3 Creating user-specific data types	75
7.3.1 Object DUT	75
7.4 Creating programming objects	78
7.4.1 Object POU	78
7.4.2 Object Action.....	87
7.4.3 Object Transition	88
7.4.4 Object Method.....	90
7.4.5 Object Property	96

7.4.6	Object Interface.....	102
7.5	Creating source code in IEC	108
7.5.1	FBD/LD/IL	108
7.5.2	Continuous Function Chart (CFC).....	113
7.5.3	Structured Text (ST), Extended Structured Text (ExST).....	123
7.5.4	Sequential Function Chart (SFC).....	124
7.5.5	Ladder Diagram (LD) (Beta).....	128
7.6	Creating a referenced task.....	131
7.6.1	Object Referenced Task	131
7.7	Creating a class diagram	133
7.8	Configuring the memory reserve for Online Change.....	133
7.9	Calling a function block, function or method with external implementation.....	134
7.10	Using the input wizard	135
7.11	Using pragmas	137
7.12	Managing text in a text list.....	138
7.12.1	Managing Static Text in Global Text Lists.....	142
7.12.2	Managing Dynamic Text in a Text List.....	144
7.13	Using image pools.....	146
7.13.1	Object Image Pool.....	146
7.13.2	Creating an Image Pool	147
7.14	Checking the syntax and analyzing the code.....	148
7.14.1	Checking Syntax	148
7.14.2	Code analysis (Static Analysis).....	148
7.15	Orientation and navigation	157
7.15.1	Using the Cross Reference List to find Occurrences	157
7.15.2	Finding Declarations	157
7.15.3	Set and use bookmarks	158
7.16	Find and Replace in the entire project	159
7.17	Refactoring.....	160
7.18	Data persistence	162
7.19	Using function blocks for implicit checks.....	163
7.19.1	Object POU's for implicit checks.....	163
7.20	Object-oriented programming	169
7.20.1	Object Function block.....	171
7.20.2	Object Method.....	173
7.20.3	Object Property	179
7.20.4	Object Interface.....	185
7.20.5	Extending a function block	191
7.20.6	Extending a structure	197
7.20.7	Extending interfaces.....	197
7.20.8	Implementation of an interface.....	198
7.20.9	Method call.....	199
7.20.10	ABSTRACT concept	201
7.20.11	Samples	202
8	Transfer PLC project to the PLC.....	209
8.1	Generating program code	209

8.2	Loading program code, logging in and starting the PLC	209
8.3	Loading the program automatically	210
9	Testing a PLC project and troubleshooting.....	211
9.1	Use of breakpoints	211
9.2	Stepwise processing of the program (stepping).....	213
9.3	Forcing and Writing Variables Values	214
9.4	Resetting the PLC project	218
9.5	Flow Control	219
9.6	Determining the Current Processing Position with the Call Stack.....	221
10	PLC project at runtime.....	222
10.1	Monitoring Values	222
10.1.1	Monitoring in Programming Objects.....	222
10.1.2	Using Watchlists.....	226
10.2	Changing Values with Recipes	228
10.2.1	Object Recipe Manager	228
10.2.2	Object Recipe Definition.....	232
10.2.3	Using recipes	233
10.2.4	Library Recipe Management - RecipeManCommands	235
10.3	Error analysis with core dump	247
10.4	PLC operation control	249
11	Updating the PLC project on the PLC	250
11.1	Performing an Online Change.....	251
11.2	Execution of a Download	253
12	Using a stand-alone PLC project.....	255
12.1	Creating a stand-alone PLC Project.....	255
12.2	Integrating the stand-alone PLC project into a TwinCAT project	256
12.2.1	Creating a TMC file and adding it to the TwinCAT project.....	256
12.2.2	Assigning a task.....	259
12.2.3	Loading program code, logging in and starting the PLC	260
12.3	Best practice	261
12.4	FAQ.....	265
13	Using libraries	266
13.1	Recommendations and notes	268
13.2	Library creation	269
13.2.1	Command Save as library.....	270
13.2.2	Command Save as library and install.....	271
13.3	Library installation	272
13.3.1	Library Repository	272
13.4	Library management	275
13.4.1	Library Manager.....	276
13.5	Library placeholders	279
13.5.1	Placeholder	280
13.5.2	Changing the placeholder resolution.....	281
13.6	Library documentation.....	282
13.6.1	Basics.....	283

13.6.2	Extended – reStructuredText	288
13.7	Other commands and dialogs	358
13.7.1	Command Add library	358
13.7.2	Add library without placeholder resolution command.....	359
13.7.3	Command Try reload library.....	360
13.7.4	Command Delete library	361
13.7.5	Command Details.....	361
13.7.6	Command Dependencies.....	362
13.7.7	Command Properties	362
13.7.8	Command Set to Effective Version	364
13.7.9	Command Set to Always Newest Version.....	364
14	Multi-task data access synchronization in the PLC.....	366
14.1	Mutex procedure (TestAndSet, FB_!ecCriticalSection) for securing critical sections	367
14.1.1	TestAndSet	370
14.1.2	FB_!ecCriticalSection.....	372
14.2	Data exchange via the PLC process image	373
14.3	Data exchange via synchronized buffers	374
15	Creating a visualization	376
15.1	Visualization Editor.....	376
15.1.1	Interface editor	378
15.1.2	Hotkeys configuration.....	382
15.1.3	Element List Editor	383
15.1.4	Toolbox	384
15.1.5	Properties window.....	385
15.2	Visualization Manager	387
15.2.1	Settings	388
15.2.2	Dialog settings.....	390
15.2.3	Default Hotkeys.....	392
15.2.4	Visualizations	393
15.2.5	User management.....	393
15.2.6	Font.....	400
15.3	Visualization Libraries	401
15.4	Preconditions	402
15.5	PLC project properties	402
15.6	Visualization Profiles	402
15.7	Visualization object	402
15.8	Visualization elements	404
15.8.1	General configuration.....	405
15.8.2	Common Controls	421
15.8.3	Basic	475
15.8.4	Lamps/Switches/Bitmaps	527
15.8.5	Measurement controls.....	537
15.8.6	Special controls.....	580
15.9	Visualization variants	602
15.9.1	Integrated visualization	602

15.9.2	PLC HMI.....	603
15.9.3	PLC HMI Web	608
15.9.4	Availability	610
15.10	Application tips	612
15.10.1	Handling of visualization pages.....	612
15.10.2	Text and language	614
15.10.3	Images	620
15.10.4	Keyboard operation in online mode	621
16	Reference Programming	622
16.1	Programming languages and their editors	622
16.1.1	Declaration Editor.....	622
16.1.2	Common Functions in Graphical Editors.....	623
16.1.3	Structured Text and Extended Structured Text (ExST).....	624
16.1.4	Sequential Function Chart (SFC).....	635
16.1.5	Function Block Diagram / Ladder / Instruction List (FBD/LD/IL)	651
16.1.6	Continuous Function Chart (CFC) and Page-Oriented CFC.....	666
16.1.7	Ladder Diagram (LD) (Beta).....	680
16.2	Variables	682
16.2.1	Local Variables - VAR	683
16.2.2	Input Variables - VAR_INPUT	683
16.2.3	Output Variables - VAR_OUTPUT	683
16.2.4	Input/Output Variables - VAR_IN_OUT, VAR_IN_OUT CONSTANT	684
16.2.5	Global Variables - VAR_GLOBAL	687
16.2.6	Temporary Variable - VAR_TEMP	687
16.2.7	Static Variables - VAR_STAT	687
16.2.8	External Variables - VAR_EXTERNAL	688
16.2.9	Instance Variables - VAR_INST	688
16.2.10	Constant variables - CONSTANT	689
16.2.11	Generic constant variables - VAR_GENERIC CONSTANT	689
16.2.12	Remanent Variables - PERSISTENT, RETAIN.....	691
16.2.13	SUPER.....	692
16.2.14	THIS	694
16.2.15	Variable types - attribute keywords	695
16.3	Operators	696
16.3.1	Address operators.....	700
16.3.2	Arithmetic operators	701
16.3.3	Call operators.....	707
16.3.4	Selection operators	708
16.3.5	Bitshift operators	710
16.3.6	Bitstring operators	712
16.3.7	Namespace operators.....	715
16.3.8	Numeric operators.....	716
16.3.9	Type conversion operators.....	721
16.3.10	Comparison operators.....	730
16.3.11	Further operators	732
16.4	Operands	744

16.4.1	BOOL Constants	745
16.4.2	Numeric Constants.....	745
16.4.3	REAL/LREAL Constants	746
16.4.4	STRING Constants	747
16.4.5	TIME/LTIME constants.....	747
16.4.6	Date and time constants	749
16.4.7	Typed Literals.....	751
16.4.8	Access to Variables in Arrays, Structures, and Blocks	751
16.4.9	Bit Access to Variables	752
16.4.10	Addresses	754
16.4.11	Functions.....	756
16.5	Data types	756
16.5.1	BOOL	758
16.5.2	Integer Data Types.....	758
16.5.3	Subrange Types.....	758
16.5.4	BIT.....	759
16.5.5	REAL/LREAL	759
16.5.6	STRING.....	760
16.5.7	WSTRING	760
16.5.8	TIME/LTIME	760
16.5.9	Date and time data types	761
16.5.10	ANY and ANY_<type>	762
16.5.11	Special data types XINT, UXINT, XWORD and PVOID.....	767
16.5.12	POINTER	767
16.5.13	Data type __SYSTEM.ExceptionCode.....	769
16.5.14	Interface pointer / INTERFACE.....	770
16.5.15	REFERENCE	770
16.5.16	ARRAY	773
16.5.17	Structure.....	778
16.5.18	Enumerations	781
16.5.19	Alias	784
16.5.20	UNION.....	784
16.6	Global data types	788
16.6.1	Overview	788
16.6.2	PlcAppSystemInfo.....	788
16.6.3	PlcTaskSystemInfo	790
16.6.4	ST_LibVersion.....	791
16.7	Alignment	791
16.8	Pragmas.....	793
16.8.1	Message pragmas.....	793
16.8.2	Attribute pragmas.....	794
16.8.3	Conditional pragmas	837
16.8.4	Region pragma.....	843
16.8.5	Pragmas for warning suppression.....	843
16.9	Identifier	844
16.10	Shading rules	845

16.11	Keywords	847
16.12	Methods FB_init, FB_reinit and FB_exit.....	848
16.12.1	FB_init.....	850
16.12.2	FB_reinit.....	853
16.12.3	FB_exit.....	854
16.12.4	Operating cases.....	855
16.12.5	Behavior with derived function blocks.....	859
17	Reference User Interface.....	861
17.1	File	861
17.1.1	Archiving options.....	861
17.1.2	Command Project... (Create new TwinCAT project).....	867
17.1.3	Command Project/Solution (Open Project/Solution).....	868
17.1.4	Command Open Project from Target.....	869
17.1.5	Command New Project... (Add new TwinCAT project).....	869
17.1.6	Command Existing Item... (Add existing TwinCAT project).....	869
17.1.7	Command Recent Projects and Solutions	870
17.1.8	Command Save All	870
17.1.9	Command Save	870
17.1.10	Command Save <Solution name> as	870
17.1.11	Command Save <TwinCAT project name> as.....	870
17.1.12	Command Save <PLC project name> as.....	871
17.1.13	Command Create disassembly file	871
17.1.14	Command Send by E-Mail.....	871
17.1.15	Command Close Solution	871
17.1.16	Command Close	871
17.1.17	Command Exit.....	872
17.1.18	Command Page Setup.....	872
17.1.19	Command Print	872
17.2	Edit.....	873
17.2.1	Standard Commands	873
17.2.2	Command Delete	873
17.2.3	Command Select All.....	873
17.2.4	Command Input Assistant.....	873
17.2.5	Command Auto Declare.....	875
17.2.6	Command Add to Watch.....	879
17.2.7	Command Browse Call Tree	879
17.2.8	Command Go To.....	879
17.2.9	Command Go To Definition.....	880
17.2.10	Command Go to Instance	880
17.2.11	Command Go to implementation	880
17.2.12	Command Go to reference.....	880
17.2.13	Command Find all references	881
17.2.14	Command Navigate To	881
17.2.15	Command Make Uppercase	881
17.2.16	Command Make Lowercase	881
17.2.17	Command View white spaces.....	881

17.2.18	Command Comment Selection	882
17.2.19	Command Uncomment Selection	882
17.2.20	Command Quick Find	882
17.2.21	Command Quick Replace	884
17.2.22	Command Switch write mode	885
17.2.23	Command Rename	886
17.2.24	Command Edit object (offline).....	886
17.2.25	Command Rename '<variable>'	886
17.2.26	Command Add '<variable>'	887
17.2.27	Command Remove '<variable>'	889
17.2.28	Command Reorder variables	889
17.3	View	890
17.3.1	Command Open object	890
17.3.2	Command Textual view.....	891
17.3.3	Command Tabular view	891
17.3.4	Command Full screen	891
17.3.5	Command Toolbars.....	891
17.3.6	Command Solution Explorer	892
17.3.7	Command Properties Window	892
17.3.8	Command Toolbox.....	893
17.3.9	Command Error List.....	894
17.3.10	Command Output.....	895
17.4	Project.....	896
17.4.1	Command Add New Item (project).....	896
17.4.2	Command Add Existing Item (Project).....	897
17.4.3	Command Properties (object)	901
17.4.4	Command Properties (PLC project).....	906
17.4.5	PLC project settings	924
17.5	Build	928
17.5.1	Command Build Solution.....	928
17.5.2	Command Rebuild Solution	928
17.5.3	Command Clean Solution	928
17.5.4	Command Check all objects	928
17.5.5	Command Build TwinCAT project.....	929
17.5.6	Command Rebuild a TwinCAT project.....	929
17.5.7	Command Clean TwinCAT project.....	929
17.5.8	Command Build PLC project.....	929
17.5.9	Command Rebuild a PLC project.....	930
17.5.10	Command Clean PLC project	930
17.6	Debug.....	930
17.6.1	Command New Breakpoint	930
17.6.2	Command Edit Breakpoint	934
17.6.3	Command Enable Breakpoint	934
17.6.4	Command Disable Breakpoint	934
17.6.5	Command Toggle Breakpoint	934
17.6.6	Command Step over	935

17.6.7	Command Step into	935
17.6.8	Command Step out	935
17.6.9	Command Run To Cursor	936
17.6.10	Command Show Next Statement	936
17.6.11	Command Set next statement	936
17.7	TwinCAT	937
17.7.1	Command Activate configuration	937
17.7.2	Command Restart TwinCAT System	937
17.7.3	Command Restart TwinCAT (Config mode)	937
17.7.4	Command Reload Devices	937
17.7.5	Command Scan	937
17.7.6	Command Toggle Free Run State	938
17.7.7	Command Show Online Data	938
17.7.8	Command Choose Target System	938
17.7.9	Command Show Sub Items	938
17.7.10	Command Software Protection	939
17.7.11	Command Hide Disabled Items	939
17.8	PLC	939
17.8.1	Window	939
17.8.2	Core dump	951
17.8.3	PLC Bookmarks	953
17.8.4	Command Download	955
17.8.5	Command Online Change	955
17.8.6	Command Login	957
17.8.7	Command Start	958
17.8.8	Command Stop	959
17.8.9	Command Logout	959
17.8.10	Command Reset cold	959
17.8.11	Command Reset origin	959
17.8.12	Command Single cycle	960
17.8.13	Command Flow Control	960
17.8.14	Command Force values	960
17.8.15	Command Unforce values	962
17.8.16	Command Write values	963
17.8.17	Command Display Mode - Binary, Decimal, Hexadecimal	963
17.8.18	Command Presentation of inheritance - Simple, Structured	964
17.8.19	Command Create Localization Template	964
17.8.20	Command Manage Localization	965
17.8.21	Command Toggle Localization	965
17.8.22	Command Active PLC project	965
17.8.23	Command Active PLC instance	966
17.9	Tools	966
17.9.1	Command Options	966
17.9.2	Command Customize	995
17.10	Window	998
17.10.1	Command Float	998

17.10.2	Command Dock	998
17.10.3	Command Hide	998
17.10.4	Command Auto Hide All	999
17.10.5	Command Auto Hide	999
17.10.6	Command Pin tab	999
17.10.7	Command New Horizontal Tab Group	1000
17.10.8	Command New Vertical Tab Group	1000
17.10.9	Command Reset Window-Layout	1000
17.10.10	Command Close All Documents	1000
17.10.11	Command Window	1000
17.10.12	Window submenu commands	1001
17.11	SFC	1001
17.11.1	Command Init step	1001
17.11.2	Command Insert step transition	1001
17.11.3	Command Insert step-transition after	1002
17.11.4	Command Parallel	1002
17.11.5	Command Alternative	1002
17.11.6	Command Insert Branch	1003
17.11.7	Command Insert branch right	1003
17.11.8	Command Insert action association	1004
17.11.9	Command Insert action association after	1005
17.11.10	Command Insert jump	1005
17.11.11	Command Insert jump after	1006
17.11.12	Command Insert macro	1006
17.11.13	Command Insert macro after	1006
17.11.14	Command Show macro	1007
17.11.15	Command Exit macro	1007
17.11.16	Command Insert after	1007
17.11.17	Command Add entry action	1007
17.11.18	Command Add exit action	1008
17.11.19	Command Change duplication - Set	1009
17.11.20	Command Change duplication - Remove	1009
17.11.21	Command Insert step	1009
17.11.22	Command Insert step after	1010
17.11.23	Command Insert transition	1010
17.11.24	Command Insert transition after	1010
17.12	CFC	1011
17.12.1	Command Edit Worksheet	1011
17.12.2	Command Edit page size	1011
17.12.3	Command Negate	1012
17.12.4	Command EN/ENO	1012
17.12.5	Command None	1012
17.12.6	Command R (reset)	1013
17.12.7	Command S (set)	1013
17.12.8	Command REF = (reference assignment)	1013
17.12.9	Command Display Execution Order	1014

17.12.10 Command Set Start of Feedback.....	1014
17.12.11 Command Move to Beginning.....	1015
17.12.12 Command Move to End	1015
17.12.13 Command Forward by one.....	1016
17.12.14 Command Back by one.....	1016
17.12.15 Command Set Execution Order	1016
17.12.16 Command Order by Data Flow	1017
17.12.17 Command Order By Topology	1017
17.12.18 Command Connect Selected Pins	1017
17.12.19 Command Unlock Connection	1018
17.12.20 Command Show Next Collision.....	1018
17.12.21 Command Select Connected Pins	1018
17.12.22 Command Use Attributed Component as Input	1019
17.12.23 Command Reset Pins	1019
17.12.24 Command Remove Unused Pins.....	1020
17.12.25 Command Add Input Pin	1020
17.12.26 Command Add Output Pin	1020
17.12.27 Command Route All Connections.....	1021
17.12.28 Command Create Control Point.....	1021
17.12.29 Command Remove Control Point	1021
17.12.30 Command Connection Mark	1022
17.12.31 Command Create group.....	1022
17.12.32 Command Ungroup.....	1022
17.12.33 Command Edit Parameters.....	1023
17.12.34 Command Force FB input.....	1024
17.12.35 Command Save prepared parameters in the project.....	1024
17.13 FBD/LD/IL	1025
17.13.1 Command Insert Contact (right).....	1025
17.13.2 Command Insert Network	1025
17.13.3 Command Insert Network (below).....	1025
17.13.4 Command Toggle comment state	1026
17.13.5 Command Insert Assignment.....	1026
17.13.6 Command Insert Box	1026
17.13.7 Command Insert Box with EN/ENO	1027
17.13.8 Command Insert Empty Box	1027
17.13.9 Command Insert Box with EN/ENO	1027
17.13.10 Command Insert jump.....	1027
17.13.11 Command Insert label	1028
17.13.12 Command Insert Return.....	1028
17.13.13 Command Insert Input.....	1028
17.13.14 Command Insert box in parallel (below).....	1029
17.13.15 Command Insert Coil	1029
17.13.16 Command Insert Set coil.....	1029
17.13.17 Command Insert Reset coil.....	1029
17.13.18 Command Insert Contact	1030
17.13.19 Command Insert Contact Parallel (below)	1030

17.13.20	Command Insert Contact Parallel (above)	1030
17.13.21	Command Insert Negated Contact	1031
17.13.22	Command Insert Negated Contact Parallel (below)	1031
17.13.23	Command Paste Contacts: Paste below	1031
17.13.24	Command Paste Contacts: Paste above	1031
17.13.25	Command Paste Contacts: Paste right (after)	1032
17.13.26	Command Insert IL line below	1032
17.13.27	Command Delete IL line	1032
17.13.28	Command Negation	1032
17.13.29	Command Edge detection	1033
17.13.30	Command Set/Reset	1033
17.13.31	Command Set output connection	1033
17.13.32	Command Insert Branch	1034
17.13.33	Command Insert Branch above	1034
17.13.34	Command Insert Branch below	1034
17.13.35	Command Set Branch Start Point	1034
17.13.36	Command Set Branch End Point	1035
17.13.37	Command Toggle parallel mode	1035
17.13.38	Command Update parameters	1035
17.13.39	Command Remove unused FB call parameters	1036
17.13.40	Command Repair POU	1036
17.13.41	Command View as function block diagram	1036
17.13.42	Command View as ladder logic	1037
17.13.43	Command View as instruction list	1037
17.13.44	Command Go To	1037
17.14	Ladder editor	1038
17.14.1	Command Outcommented	1038
17.14.2	Command Negate	1038
17.14.3	Command Open Parallel Branch	1038
17.14.4	Command Close Parallel Branch	1039
17.14.5	Command Set/Reset – Set, Set/Reset – Reset	1039
17.14.6	Command Edge Detection – Rising Edge	1039
17.14.7	Command Edge Detection – Falling Edge	1039
17.14.8	Command EN/ENO: EN	1039
17.14.9	Command EN/ENO: ENO	1040
17.14.10	Command Insert Network	1040
17.14.11	Command Insert Contact	1040
17.14.12	Command Insert Coil	1041
17.14.13	Command Insert Box	1041
17.14.14	Command Insert Jump	1042
17.14.15	Command Insert Return	1042
17.14.16	Command Insert Input	1042
17.14.17	Command Insert Output	1043
17.14.18	Command Convert to New Ladder	1043
17.15	Declarations	1043
17.15.1	Command Paste	1043

17.15.2	Command Edit the declaration header.....	1043
17.15.3	Command Move Down.....	1044
17.15.4	Command Move Up	1045
17.16	Textlist.....	1045
17.16.1	Command Add Language	1045
17.16.2	Command Remove Language	1045
17.16.3	Command Insert Text.....	1046
17.16.4	Command Import/Export Text Lists.....	1046
17.16.5	Command Remove Unused Text List Records.....	1047
17.16.6	Command Check Visualization Text Ids	1047
17.16.7	Command Update Visualization Text Ids	1048
17.16.8	Command Export All	1048
17.16.9	Command Export All Unicode	1048
17.16.10	Command Add text list support	1049
17.16.11	Command Remove text list support.....	1049
17.17	Recipes	1049
17.17.1	Command Add a new recipe.....	1049
17.17.2	Command Remove recipe	1050
17.17.3	Command Load Recipe	1050
17.17.4	Command Save recipe.....	1051
17.17.5	Command Read Recipe.....	1051
17.17.6	Command Write Recipe.....	1051
17.17.7	Command Load and Write Recipe	1052
17.17.8	Command Read and Save Recipe.....	1052
17.17.9	Command Insert variable	1052
17.17.10	Command Remove variables.....	1053
17.17.11	Command Update structured variables.....	1053
17.17.12	Command Download recipes from the device	1054
17.18	Library	1055
17.19	Visualization	1055
17.19.1	Command Interface Editor	1056
17.19.2	Command Hotkey Configuration.....	1056
17.19.3	Command Element List.....	1056
17.19.4	Command Align Left.....	1056
17.19.5	Command Align Top	1056
17.19.6	Command Align Right	1056
17.19.7	Command Align Bottom	1056
17.19.8	Command Align Vertical Center.....	1057
17.19.9	Command Align Horizontal Center.....	1057
17.19.10	Command Make horizontal spacing equal	1057
17.19.11	Command Increase horizontal spacing.....	1057
17.19.12	Command Decrease horizontal spacing	1057
17.19.13	Command Remove horizontal spacing	1058
17.19.14	Command Make vertical spacing equal	1058
17.19.15	Command Increase vertical spacing	1058
17.19.16	Command Decrease vertical spacing	1058

17.19.17 Command Remove vertical spacing	1059
17.19.18 Command Make same width.....	1059
17.19.19 Command Make same height	1059
17.19.20 Command Make same size.....	1059
17.19.21 Command Size to Grid.....	1059
17.19.22 Command Bring One to Front.....	1060
17.19.23 Command Bring to front.....	1060
17.19.24 Command Send One to Back	1060
17.19.25 Command Send to Back	1060
17.19.26 Command Group.....	1060
17.19.27 Command Ungroup.....	1061
17.19.28 Command Background	1061
17.19.29 Command Select All.....	1061
17.19.30 Command Deselect All.....	1062
17.19.31 Command Multiply visu element	1062
17.19.32 Command Activate keyboard usage	1063
17.20 Miscellaneous	1063
17.20.1 Command Implement interfaces	1063
17.21 Context menu TwinCAT project	1064
17.21.1 Command Save <TwinCAT project name> as Archive.....	1064
17.21.2 Command Send by E-Mail.....	1064
17.21.3 Command Backup <TwinCAT project names> automatically to the target system	1065
17.21.4 Command Compare <TwinCAT project name> with the target system.....	1065
17.21.5 Command Update project with target system.....	1065
17.21.6 Command Load project with TwinCAT 2.xx Version.....	1065
17.21.7 Command Show Hidden Configurations	1065
17.21.8 Command Remove From Solution.....	1066
17.21.9 Command Rename	1066
17.21.10 Command Build TwinCAT project.....	1066
17.21.11 Command Rebuild a TwinCAT project.....	1066
17.21.12 Command Clean TwinCAT project.....	1066
17.21.13 Command Unload Project	1067
17.21.14 Import AutomationML via AML DataExchange.....	1067
17.21.15 Export AutomationML.....	1067
18 PLC programming conventions.....	1068
18.1 Programming style	1070
18.1.1 Font and editor settings.....	1070
18.1.2 Language	1071
18.1.3 Project structure	1071
18.1.4 Program structure	1073
18.2 Naming conventions.....	1080
18.2.1 General	1080
18.2.2 Identifier	1083
18.2.3 Global variable lists and parameter lists	1087
18.2.4 Samples	1088
18.3 Programming.....	1089

18.3.1	General	1091
18.3.2	Libraries	1100
18.3.3	DUTs	1101
18.3.4	POUs.....	1104
18.3.5	Variables	1114
18.3.6	Runtime behavior	1125
19	Samples	1129
19.1	Basic samples	1129
19.1.1	First steps – state machine, timer, trigger	1129
19.1.2	First steps – basic PLC elements.....	1129
19.1.3	STRING functions	1130
19.1.4	OOP basic sample	1130
19.2	Extended samples.....	1130
19.2.1	OOP extended sample.....	1130
19.2.2	Byte-Alignment.....	1131
19.2.3	Multi-task data access synchronization.....	1131
19.2.4	Library documentation reStructuredText.....	1131

1 Foreword

1.1 Notes on the documentation

This description is intended exclusively for trained specialists in control and automation technology who are familiar with the applicable national standards.

For installation and commissioning of the components, it is absolutely necessary to observe the documentation and the following notes and explanations.

The qualified personnel is obliged to always use the currently valid documentation.

The responsible staff must ensure that the application or use of the products described satisfies all requirements for safety, including all the relevant laws, regulations, guidelines, and standards.

Disclaimer

The documentation has been prepared with care. The products described are, however, constantly under development.

We reserve the right to revise and change the documentation at any time and without notice.

No claims to modify products that have already been supplied may be made on the basis of the data, diagrams, and descriptions in this documentation.

Trademarks

Beckhoff®, TwinCAT®, TwinCAT/BSD®, TC/BSD®, EtherCAT®, EtherCAT G®, EtherCAT G10®, EtherCAT P®, Safety over EtherCAT®, TwinSAFE®, XFC®, XTS® and XPlanar® are registered and licensed trademarks of Beckhoff Automation GmbH.

If third parties make use of designations or trademarks used in this publication for their own purposes, this could infringe upon the rights of the owners of the said designations.

Patents

The EtherCAT Technology is covered, including but not limited to the following patent applications and patents:

EP1590927, EP1789857, EP1456722, EP2137893, DE102015105702
and similar applications and registrations in several other countries.

EtherCAT®

EtherCAT® is registered trademark and patented technology, licensed by Beckhoff Automation GmbH, Germany

Copyright

© Beckhoff Automation GmbH & Co. KG, Germany.

The distribution and reproduction of this document as well as the use and communication of its contents without express authorization are prohibited.

Offenders will be held liable for the payment of damages. All rights reserved in the event that a patent, utility model, or design are registered.

1.2 For your safety

Safety regulations

Read the following explanations for your safety.

Always observe and follow product-specific safety instructions, which you may find at the appropriate places in this document.

Exclusion of liability

All the components are supplied in particular hardware and software configurations which are appropriate for the application. Modifications to hardware or software configurations other than those described in the documentation are not permitted, and nullify the liability of Beckhoff Automation GmbH & Co. KG.

Personnel qualification

This description is only intended for trained specialists in control, automation, and drive technology who are familiar with the applicable national standards.

Signal words

The signal words used in the documentation are classified below. In order to prevent injury and damage to persons and property, read and follow the safety and warning notices.

Personal injury warnings**⚠ DANGER**

Hazard with high risk of death or serious injury.

⚠ WARNING

Hazard with medium risk of death or serious injury.

⚠ CAUTION

There is a low-risk hazard that could result in medium or minor injury.

Warning of damage to property or environment**NOTICE**

The environment, equipment, or data may be damaged.

Information on handling the product

This information includes, for example:
recommendations for action, assistance or further information on the product.

1.3 Notes on information security

The products of Beckhoff Automation GmbH & Co. KG (Beckhoff), insofar as they can be accessed online, are equipped with security functions that support the secure operation of plants, systems, machines and networks. Despite the security functions, the creation, implementation and constant updating of a holistic security concept for the operation are necessary to protect the respective plant, system, machine and networks against cyber threats. The products sold by Beckhoff are only part of the overall security concept. The customer is responsible for preventing unauthorized access by third parties to its equipment, systems, machines and networks. The latter should be connected to the corporate network or the Internet only if appropriate protective measures have been set up.

In addition, the recommendations from Beckhoff regarding appropriate protective measures should be observed. Further information regarding information security and industrial security can be found in our <https://www.beckhoff.com/secguide>.

Beckhoff products and solutions undergo continuous further development. This also applies to security functions. In light of this continuous further development, Beckhoff expressly recommends that the products are kept up to date at all times and that updates are installed for the products once they have been made available. Using outdated or unsupported product versions can increase the risk of cyber threats.

To stay informed about information security for Beckhoff products, subscribe to the RSS feed at <https://www.beckhoff.com/secinfo>.

2 Quickstart

System overview

TwinCAT 3 PLC realises one or more PLCs with the international standard IEC 61131-3 3rd edition on one CPU. All programming languages described in the standard can be used for programming. The blocks of the type PROGRAM can be linked with real-time tasks. Various convenient debugging options facilitate fault-finding and commissioning. Program modifications can be carried out at any times and in any size online, i.e. when the PLC is running. All variables are available symbolically by ADS and can be read and written in appropriate clients.

Functionalities

TwinCAT 3 PLC provides you with versatile and comfortable engineering functions for your development work:

	See also in this online help:
Project configuration through wizards	Creating and configuring a project [► 54]
Interface customization	Customize interface
Generation of professional control programs according to IEC 61131-3 with many standard features	Programming a PLC project [► 66]
Convenient programming with mouse and keyboard in all IEC 61131-3 languages Corresponding editors for FBD, LD, IL, ST, SFC, plus the variants CFC and Extended CFC	Programming languages and their editors [► 622]
Input support for input and configuration of a wide range of data	Using the input wizard [► 135]
Support of object-oriented programming Genuine object-oriented programming according to 61131-3 third edition, in all IEC 61131-3 languages possible without additional tools Inheritance of program blocks to similar program parts to reduce development time and errors Object-oriented programming is not a must: functional or object-oriented programming can be used and mixed as required	Object-oriented programming [► 169]
Comprehensive project comparison, also for graphic editors	Compare projects
Library concept for convenient reuse of program code	Using libraries [► 266]
Debugging and online features to optimize program code and speed up testing and commissioning	Testing and debugging [► 211]
Integrated compilers for various CPU platforms	Project properties – Category Compile [► 910]
Security properties for protecting the source code and operating the controller	Protecting and saving a project Encrypting a PLC project

Using Help

Each help component consists of a conceptual part and a reference part. The conceptual part provides detailed explanations of all the topics that are relevant for creating, managing and executing a TwinCAT 3 PLC project. The descriptions are supplemented by step-by-step instructions for achieving the desired result. The reference parts are complete reference works for the user interface and programming of TwinCAT 3 PLC. See also the "TC3 User Interface" documentation.

2.1 Differences compared with TwinCAT 2

Library versions

- Several versions of a library are possible in the same project. Unambiguous accesses are ensured by specifying the namespace.
- Installation in repositories (databases with libraries in various versions)
- Automatic updating
- Debugging possible
- Library profiles and placeholders facilitate version compatibility

Compatibility with projects in other file formats

- The command **Add Existing Item ...** can be used to automatically convert projects with different formats to the TwinCAT 3 format. You can define how integrated libraries should be handled.
- A converter for the TwinCAT 2 format is part of the standard programming system. However, if you want to use a TwinCAT 2 project in TwinCAT 3, you have to take into account some prerequisites and restrictions.

Data types

The following data types are new:

- any_type
- UNION
- LTIME
- BIT
- References
- Enumerations: Base data type can be specified.
- `di : DINT := DINT#16#FFFFFFFF` : not permitted

Editors

ST editor:

- Bracketing, breaks, code completion, inline monitoring, inline set/reset assignment

FBD/LD/IL editor:

- FBD, LD and IL can be converted to each other and have a common editor.
- IL editor as table editor
- The main output in function blocks with multiple outputs can be set.
- Branches and “networks in networks”

SFC editor:

- Only one step type, macros, multiple selection of independent elements, no syntax check during editing

Visualization concept

- By default, the visualization editor cooperates with a toolbox and an editor for the element properties.
- Parts of the visualization functionality are implemented in accordance with IEC 61131 and are therefore made available through libraries. An internal runtime system performs the main visualization functions.
- Text lists and image pools are used for managing texts and image files.
- A visualization manager deals with different visualization clients (such as web client, standalone client...), which can be used flexibly and with little effort for a wide range of custom PLC projects.
- Visualization profiles define which library versions and which elements are currently available.
- Visualization styles facilitate adaptation of the “look and feel” of the visualizations.

Operators and variables

- New validity range operators, extended namespaces
- Init method replaces the INI operator
- Exit method
- Output variables in function and method calls
- VAR_TEMP, VAR_STAT
- Any expressions for variable initialization
- Assignment as expression
- Index access with pointers and strings

Object orientation

- Extensions for function blocks: properties, interfaces, methods, inheritance, method call

Further parameters

- Configurable menus, toolbar and keyboard operation
- Unicode support
- Single-line comments: // Comment
- CONTINUE in loops
- Conditional compilation
- Conditional breakpoints
- Debug: Run To Cursor, Step Out

Changes from TwinCAT 2 PLC:

- FUNCTIONBLOCK is no longer a valid keyword for function blocks, in place of FUNCTION_BLOCK
- TYPE (declaration of a structure) must be followed by a “.”.
- ARRAY initialization must be enclosed in parentheses.
- INI is no longer supported and must be replaced by the Init method.
- In function calls, it is no longer possible to mix explicit parameter assignments with implicit assignments. The order of the parameter input assignments can therefore be changed:

```
fun(formal1 := actual1, actual2); // → Fehlermeldung
fun(formal2 := actual2, formal1 := actual1); // gleiche Semantik wie ...
fun(formal1 := actual1, formal2 := actual2);
```

- Pragmas (import of TwinCAT 2 pragmas is not yet implemented)
- The TRUNC operator now converts to the data type DINT instead of INT; during a TwinCAT 2 import, a corresponding type conversion is added automatically.
- Direct addressing of allocated variables:

The direct [addressing \[▶ 754\]](#) of allocated variables has changed. Whereas the starting point was always the same with TwinCAT 2 irrespective of the data type, a distinction is made with TwinCAT 3:

TwinCAT 2: W0 contains B0 and B1, W1 contains B1 and B2, W100 contains B100 and B101

TwinCAT 3: W0 contains B0 and B1, W1 contains B2 and B3, W100 contains B200 and B201

2.2 Your first TwinCAT 3 PLC project

Contents of your first project

In this tutorial, you will program a simple refrigerator control.

- As with a conventional refrigerator, the set temperature is set by the user via a control knob.
- The refrigerator detects the actual temperature via a sensor. If this is too high, the refrigerator starts the compressor with an adjustable delay.

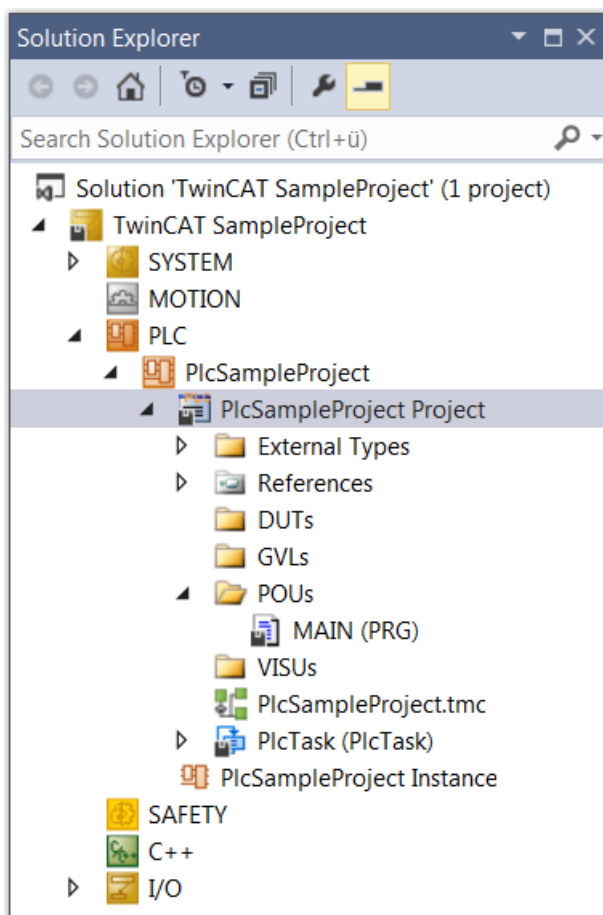
- The compressor cools until the set temperature minus a hysteresis of 1 degree is reached. The hysteresis is intended to prevent the actual temperature from oscillating too much around the set temperature, and to prevent the compressor from continuously switching on and off.
- When the door is open, a lamp lights up inside the refrigerator.
- If the door is open for too long, a timed acoustic signal will sound.
- If the compressor does not reach the set temperature for an extended period of time despite the motor's activity, the beeper emits a continuous acoustic signal.

Project planning:

The cooling activity is controlled in the main program of the PLC project, the signal management take place in another program block. The required standard function blocks are available in the Tc2_Standard library. Since no real temperature sensors and no real actuators are connected in this example project, you will also write a program for simulating the rise and fall in temperature. This allows you to monitor the operation of the refrigerator control unit in online mode. You define variables that are to be used by all function blocks in a global variable list.

Creating the PLC project

1. In the **File** menu, select **New > Project** to create a new TwinCAT project file.
 - ⇒ A new Solution with the TwinCAT project tree opens in the **Solution Explorer**.
2. In the **Solution Explorer** select the **PLC** node and the command **Add New Item**, in order to add a PLC project to the TwinCAT project.
 - ⇒ The dialog **Add New Item – TwinCAT <project name>** opens.
3. In the category **Plc Templates** select the **Standard PLC project** template.
4. Enter a name and storage location for the project and click the **Add** button.
 - ⇒ The selected template automatically creates a MAIN program, which is called by a task. "Structured Text (ST)" is automatically selected as the programming language.



The Library Manager with some important default libraries is automatically selected under


References. The Tc2_Standard library contains all the functions and function blocks described by the IEC 61131-3 standard.

- ▲ References
 - Tc2_Standard
 - Tc2_System
 - Tc3_Module

Declaring global variables

First, declare the variables that you want to use throughout the PLC project. To do this, you create a global variable list:

1. Select the subfolder **GVLs** in the PLC project tree.
2. In the context menu select the command **Add > Global Variable List**.
3. Change the automatically entered name "GVL" to "GVL_Var".
4. Confirm with **Open**.

⇒ The object "GVL_Var" () appears in the PLC project tree in the subfolder **GVLs**. The GVL editor opens.

⇒ When the textual view appears, the keywords VAR_GLOBAL and END_VAR are already included.


5. Activate the tabular view for the example by clicking on the  button in the right sidebar of the editor.

⇒ An empty row appears. The cursor is located in the column **Name**.

6. Enter "fTempActual" in the **Name** field.

⇒ At the same time, the scope VAR_GLOBAL and the data type BOOL are automatically entered in the row.

7. Double-click the field in the **Data type** column.

⇒ The field is now editable, and the button  appears.

8. Click the button and select **Input Assistant**.

⇒ The **Input Assistant** dialog opens.

9. Select the data type REAL and click **OK**.

10. Enter a numerical value in the **Initialization** column, for example "8.0".


⇒ Declare the following variables in the same way:

Name	Data type	Initialization	Comment
fTempActual	REAL	1.0	Actual temperature
fTempSet	REAL	8.0	Set temperature
bDoorOpen	BOOL	FALSE	Door status
tImAlarmThreshold	TIME	T#30s	Compressor running time after which an alarm sounds.
tDoorOpenThreshold	TIME	T#10s	Time from door opening after which an alarm sounds.
xCompressor	BOOL	FALSE	Control signal
xSignal	BOOL	FALSE	Control signal
xLamp	BOOL	FALSE	Status message

Creating the main program for cooling control in the CFC editor

In the MAIN program block created by default, you describe the main function of the PLC program: The compressor becomes active and cools when the actual temperature is higher than the set temperature plus a hysteresis. The compressor is switched off when as the actual temperature is lower than the set temperature minus the hysteresis.

Perform the following steps to describe this functionality in the implementation language "Continuous Function Chart (CFC)":

- ✓ Since the automatically created MAIN program block is created by default in the implementation language "Structured Text (ST)", you must first delete this program. With the context menu command **Add > POU...** you create a new MAIN program in the implementation language "Continuous Function Chart (CFC)".
- 1. Then double-click on the **MAIN** program in the PLC project tree (subfolder **POUs**).
 - ⇒ The CFC Editor opens with the **MAIN** tab. The Declaration editor in textual or tabular representation is displayed above the graphical editor area. On the right is the **Toolbox** view. If the toolbox does not appear, you can use the command **Toolbox** in the **View** menu to place it on the desktop.
- 2. In the **Toolbox** view, click the **Input** element and drag it with the mouse to a position in the CFC Editor.
 - ⇒ The nameless input **???** has been inserted.
- 3. In the CFC Editor, click the input **???** and open the **Input Assistant** by clicking .
- 4. Select the variable `fTempActual` from the category **Variables** at **Project > GVLs**.
- 5. Confirm the dialog with **OK** to reference the global variable `fTempActual`.
- 6. As in step 3, create a further input with the name of the global variable `rTempSet`.
- 7. Create another input.
- 8. Click **???** and replace it with the name `fHysteresis`.
 - ⇒ Since this is not the name of an already known variable, the **Auto Declare** dialog appears. The name is already included in the dialog.
- 9. Complete the fields in the **Auto Declare** dialog with the data type **REAL** and the initialization value **"1"**.
- 10. Click the **OK** button.
 - ⇒ The variable `fHysteresis` appears in the declaration editor.
- 11. Now add an addition function block: In the **Toolbox** view, click the **Function block** element and drag it with the mouse to a position in the CFC Editor.
 - ⇒ The function block appears in the CFC Editor.
- 12. Replace **???** with **ADD**.
 - ⇒ The **ADD** (Addition) function block adds all inputs that are connected to it.
- 13. Connect the input `GVL_Var.fTempSet` with the **ADD** function block: To do this, click on the output connector of the input and drag it to the upper input of the **ADD** function block.
- 14. Connect the `fHysteresis` input to the lower input of the **ADD** function block in the same way.
 - ⇒ The two inputs `fHysteresis` and `fTempSet` are now added by **ADD**.
- 15. If you want to move an element in the editor, click on a free space in the element or on the frame so that the element is selected (red frame, shaded red).
- 16. Keep the mouse button pressed and drag the element to the desired position.
- 17. Create another function block to the right of the **ADD** function block.
 - ⇒ It is intended to compare `"GVL_Var.fTempActual"` with the sum of `"GVL_Var.fTempSet"` and `fHysteresis`.
- 18. Assign the function **GT** (Greater Than) to the function block.
 - ⇒ The **GT** function block works as follows:

```
IF (oberer Eingang > unterer Eingang) THEN Ausgang := TRUE;
```
- 19. Connect the input `"GVL_Var.fTempActual"` with the upper input of the **GT** function block.
- 20. Connect the output of the **ADD** function block with the lower input of the **GT** function block.
- 21. Now create a function block to the right of the **GT** function block that starts or stops the cooling compressor depending on the input condition (Set - Reset).
- 22. Enter the name **"SR"** in the **???** field.
- 23. Close the open input field above the function block (`SR_0`) with the Enter key.
 - ⇒ The **Auto Declare** dialog appears.
- 24. Declare the variable with the name `"fbSR"` and the data type **SR**.

25. Click the **OK** button.

- ⇒ The SR function block, which is also defined in the library Tc2_Standard, determines the THEN at the output of the GT function block. The inputs SET1 and RESET appear.

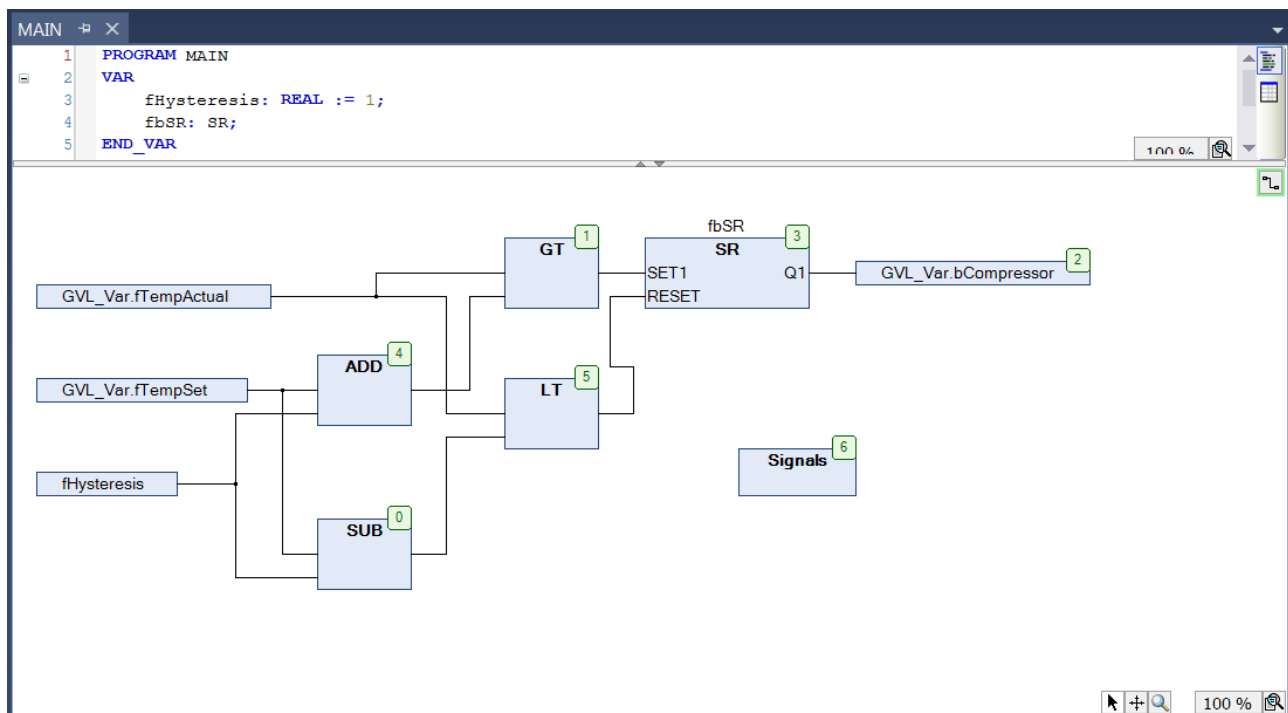
26. Connect the output connection on the right of the GT function block to the SET1 input of the fbSR function block.

- ⇒ SR can reset a Boolean variable from FALSE to TRUE. If the condition applies at input SET1, the Boolean variable is set to TRUE. If the condition applies to RESET, the variable is reset. In our example, the Boolean (global) variable is "GVL_Var.bCompressor".

27. Create an **Output** element and assign the global variable "GVL_Var.bCompressor" to it. Draw a connecting line between "GVL_Var.bCompressor" and the output connection from Q1 to SR.

Now enter the condition under which the compressor should switch off again, i.e. under which the RESET input of the SR function block receives a TRUE signal. To do this, formulate the opposite condition as described above. Use the function block SUB (Subtract) and LT (Less Than) for this purpose.

The following CFC plan is created:



Creating a program block for signal management in the Ladder Diagram editor

Now implement the signal management for the alarm buzzer and for switching the lamp on and off in a further program block. The implementation language "Ladder Diagram (LD)" is suitable for this purpose.

Use separate networks for the following signals:

- A continuous acoustic signal sounds if the compressor runs too long because the temperature is too high.
- A pulsed signal sounds if the door is open too long.
- The light is on as long as the door is open.

1. In the PLC project tree (subfolder **POUs**) create a POU object of type **Program** with the implementation language "Ladder Diagram (LD)".
2. Name the POU object "Signals".
 - ⇒ "Signals" appears in the PLC project tree below MAIN. The Ladder Diagram editor opens with the **Signals** tab. The declaration editor appears in the upper part of the screen, the **Toolbox** view on the right. The LD contains an empty network.
3. In the network, you program that an acoustic signal sounds if the cooling compressor runs for too long without reaching the set temperature. To do this, insert a TON timer function block.

- ⇒ It switches a Boolean TRUE signal to TRUE after a specified time.
- 4. Select a TON in the **Toolbox** view under **Function blocks** and drag it into the empty network to the **Start here** rectangle, which appears.
- 5. Release the mouse button when the field turns green.
 - ⇒ The function block appears as rectangle with inputs and outputs and is automatically assigned the instance name TON_0. The line editor is open and the cursor is blinking.
- 6. Confirm the instance name with Enter key.
 - ⇒ The **Auto Declare** dialog opens.
- 7. Declare the variable with the name "fbTimer1" and the data type TON.
- 8. Click the **OK** button.




If you want to read the help for the function block, mark the complete name of the function block with the cursor and press [F1].


- 9. To program that the function block is activated as soon as the cooling compressor starts running, name the contact at the upper input of the function block "GVL_Var.bCompressor".
 - ⇒ You have already defined this Boolean variable in the global variable list "GVL_Var".

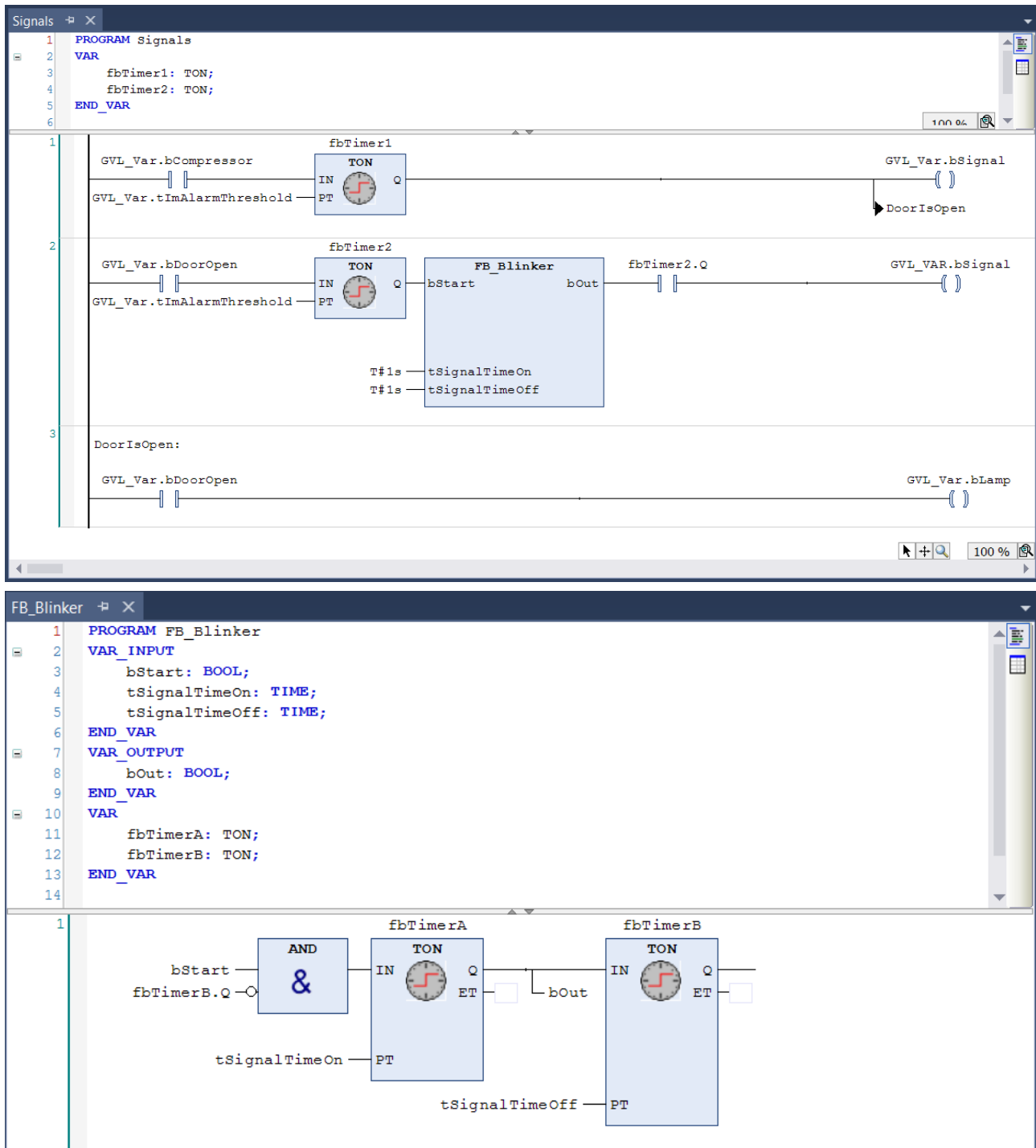


When you start entering a variable name at the input position, you will always automatically receive a list of all variables whose names begin with the characters entered and which can be used at this point. This feature is a default setting in the TwinCAT options for Smart Coding.

- 10. Insert the signal you want to activate: To do this, drag a **Coil** from the **Toolbox** view, category **Ladder Diagram elements** to output Q of the TON function block. Name the coil "GVL_Var.bSignal".
- 11. Add the variable "GVL_Var.tImAlarmThreshold" to the PT input of "fbTimer1" to define the time from activation of the TON device after which the signal should sound. To do this, click on the rectangle with a border to the right of the input connection.
- 12. Enter the variable name.
- 13. Click the TON function block and in the context menu select the command **Remove unused FB call parameters**.
 - ⇒ The unused output ET was removed.
- 14. In the second LD network, you program that the signal should sound intermittently when the door is opened too long.
- 15. First, create a new POU object of type **Function Block** in the implementation language "Function Block Diagram (FBD)" in the PLC project tree, subfolder **POUs**.
- 16. Name it "FB_Blinker".
 - ⇒ The function block FB_Blinker appears in the PLC project tree above MAIN. The FBD editor opens with the **FB_Blinker** tab. The declaration editor appears in the upper part of the screen, the **Toolbox** view on the right. The FBD contains an empty network.
 - ⇒ The flasher is to be realized by an AND operator and two TON function blocks.
- 17. Select a **function block** in the **Toolbox** view under **General** and drag it into the empty network to the **Start here** rectangle, which appears.
- 18. Release the mouse button when the field turns green.
 - ⇒ The function block appears as a rectangle with inputs and outputs.
- 19. Click ??? within the function block and enter the keyword AND in the field, which is now editable.
- 20. Confirm with Enter key.
 - ⇒ Since it is a function, no instantiation is required.
- 21. Select a TON function block in the **Toolbox** view under **Function blocks** and drag it into the network to the output of the AND function block.
- 22. Release the mouse button when the field turns green.
 - ⇒ The function block appears as rectangle with inputs and outputs and is automatically assigned the instance name TON_0.

23. Close the open input field above the function block (TON_0) with the Enter key.
 - ⇒ The **Auto Declare** dialog opens.
24. Declare the variable with the name "fbTimerA" and the data type TON.
25. Click the **OK** button.
26. Select a further TON function block in the **Toolbox** view under **Function blocks** and drag it into the network to the output of the TON function block "fbTimerA".
27. Release the mouse button when the field turns green.
 - ⇒ The function block appears as rectangle with inputs and outputs and is automatically assigned the instance name "TON_0".
28. Close the open input field above the function block (TON_0) with the Enter key.
 - ⇒ The **Auto Declare** dialog opens.
29. Declare the variable with the name "fbTimerB" and the data type TON. Click the **OK** button.
 - ⇒ The first input of the AND function block is to be connected to a Boolean variable "bStart".
30. Click ??? and enter the variable name.
31. Confirm with Enter key.
 - ⇒ The **Auto Declare** dialog opens. The name and the data type are recognized automatically.
32. Select the entry VAR_INPUT as **Scope**.
33. Confirm the dialog with **OK**.
 - ⇒ The second input of the AND function block is to be connected with the output Q of the second TON function block "fbTimerB".
34. Click ??? and open the **Input Assistant** via .
35. In the category **Variables** select the function block "fbTimerB" and the output Q.
36. Confirm the dialog with **OK**.
 - ⇒ The variable "fbTimerB.Q" is added at the second input.
37. Select the second input and open the context menu with a right-click.
38. Select the command **Negation** to negate the input.
 - ⇒ A circle appears at the corresponding input.
39. Set the time until output Q is set via the PT inputs of the TON function blocks.
40. Declare the input variables "tSignalTimeOn" and "tSignalTimeoff" for the TON function blocks "fbTimerA" and "fbTimerB" via the **Auto Declare** dialog.
41. Select the entry VAR_INPUT as **Scope**.
 - ⇒ The generated clock pulse is to be output at output Q of the TON function block "fbTimerA".
42. To do this, select an **assignment** in the **Toolbox** view under **General** and drag it into the network to the output of the TON function block "fbTimerA".
43. Release the mouse button when the field turns green.
 - ⇒ The assignment is added between the function blocks "fbTimerA" and "fbTimerB".
44. Click ??? and enter the variable name "bOut".
45. Confirm with Enter key.
 - ⇒ The **Auto Declare** dialog opens.
46. Select the **Scope** VAR_OUTPUT and the data type BOOL.
47. Confirm the dialog.
48. Finally, remove the ??? at the unused inputs and outputs of the function blocks.
 - ⇒ The completed function block FB_Blinker can now be instantiated and called.
49. Open the program "Signals" in the FBD/LD/IL editor.
50. Click below the first network in the editor window.
51. In the context menu, select the command **Insert network**.
 - ⇒ An empty network with number 2 appears.

52. As in the first network, implement a TON function block for time-controlled activation of the signal, this time triggered by the global variable "GVL_Var.bDoorOpen" at input IN.
 53. Add the global variable "GVL_Var.tImDoorOpenThreshold" at the input PT.
 54. In addition, add the function block FB_Blinker to output Q of the TON function block in this network.
 - ⇒ The function block FB_Blinker clocks the signal forwarding Q and therefore "GVL_Var.bSignal".
 55. To do this, drag a **Contact** element from the **Toolbox** view to the OUT output of the function block.
 56. Assign the variable "fbTimer2.Q" to the contact.
 57. Insert an element **Coil** after the contact and assign the global variable "GVL_Var.bSignal" to it.
 58. Assign the value T#1s to the two input variables "tSignalTimeOn" and "tSignalTimeOff" of function block FB_Blinker.
 - ⇒ The cycle time is then 1 second for TRUE and 1 second for FALSE.
 59. Click on the TON function block.
 60. Select the command **Remove unused FB call parameters** in the context menu.
 - ⇒ The unused output ET was removed.
 - ⇒ In the third network of the LD, program the lamp to light up while the door is open.
 61. To do this, add another network and at the left a contact "GVL_Var.bDoorOpen", which leads directly to an inserted coil "GVL_Var.bLamp".
 - ⇒ TwinCAT processes the networks of an LD in succession.
 62. To ensure that only network 1 or only network 2 is executed, add a jump to network 3 at the end of network 1.
 63. Select network 3 by clicking into the network or into the field with the network number.
 64. From the context menu select the command **Insert label**.
 65. Replace the text **Label:** of the label in the upper left section of the network with "DoorsOpen".
 66. Select network 1.
 67. Drag the **Jump** element into network from the **Toolbox** view, **General** category.
 68. Position it on the **Output** rectangle that appears, or at **Insert jump here**.
 - ⇒ The jump element appears. The jump destination is still shown as ???.
 69. Selecting ??? and click .
 70. Select "DoorsOpen" from the possible label identifiers.
 71. Confirm with **OK**.
 - ⇒ The label to network 3 is implemented.
- ⇒ The LD program now looks as follows:



Calling the “Signals” program in the main program

In our sample program, the MAIN program should call the “Signals” program for signal processing.

1. In the PLC project tree double-click on the MAIN program.
 - ⇒ The MAIN program opens in the editor.
2. Drag a **Box** element from the **Toolbox** view into the editor of MAIN.
3. Use the **Input Assistant** to add the "Signals" program call to this function block from the **Module Calls** category.

Creating an ST program block for a simulation

Since this sample project is not linked to real sensors and actuators, write a program for simulating the rise and fall in temperature. This allows you to monitor the operation of the refrigerator control unit in online mode.

Use "Structured Text (ST)" to create the simulation program.

The program increases the temperature until the MAIN program detects that the set temperature has been exceeded and activates the cooling compressor. The simulation program then lowers the temperature again until the main program deactivates the compressor.

1. Add a POU function block called "Simulation" of type **Program** and written in the implementation language "Structured Text (ST)" to the PLC project tree.
2. Implement the following in the ST editor:

```
PROGRAM Simulation
VAR
  fbT1          : TON;          //
  The temperature is decreased on a time delay, when the compressor has been activated
  tCooling      : TIME := T#500MS;
  bReduceTemp   : BOOL;        //Signal for decreasing the temperature
  fbT2          : TON;          //
  The temperature is increased on a time delay, when the compressor has been activated
  tEnvironment  : TIME := T#2S; //Delay time when the door is closed
  tEnvironmentDoorOpen : TIME := T#1s; //Delay time when the door is open
  bRaiseTemp    : BOOL;        //Signal for increasing the temperature
  tImTemp       : TIME;        //Delay time
  nCounter      : INT;
END_VAR

// After the compressor has been activated due to fTempActual being too high, the temperature decreases.
// The temperature is decremented by 0.1°C per cycle after a delay of P_Cooling
IF GVL_VAR.bCompressor THEN
  fbT1(IN:= GVL_Var.bCompressor, PT:= tCooling, Q=>bReduceTemp);
  IF bReduceTemp THEN
    GVL_Var.fTempActual := GVL_Var.fTempActual-0.1;
    fbT1(IN:=FALSE);
  END_IF
END_IF

//If the door is open, the warming will occur faster; SEL selects tEnvironmentDoorOpen
tImTemp:=SEL(GVL_Var.bDoorOpen, tEnvironment, tEnvironmentDoorOpen);

//If the compressor is not in operation, then the cooling chamber will become warmer.
//The temperature is incremented by 0.1°C per cycle after a delay of tImTemp
fbT2(IN:= TRUE, PT:= tImTemp, Q=>bRaiseTemp);
IF bRaiseTemp THEN
  GVL_Var.fTempActual := GVL_Var.fTempActual + 0.1;
  fbT2(IN:=FALSE);
END_IF

nCounter := nCounter+1; // No function, just for demonstration purposes.
```

● Visualization

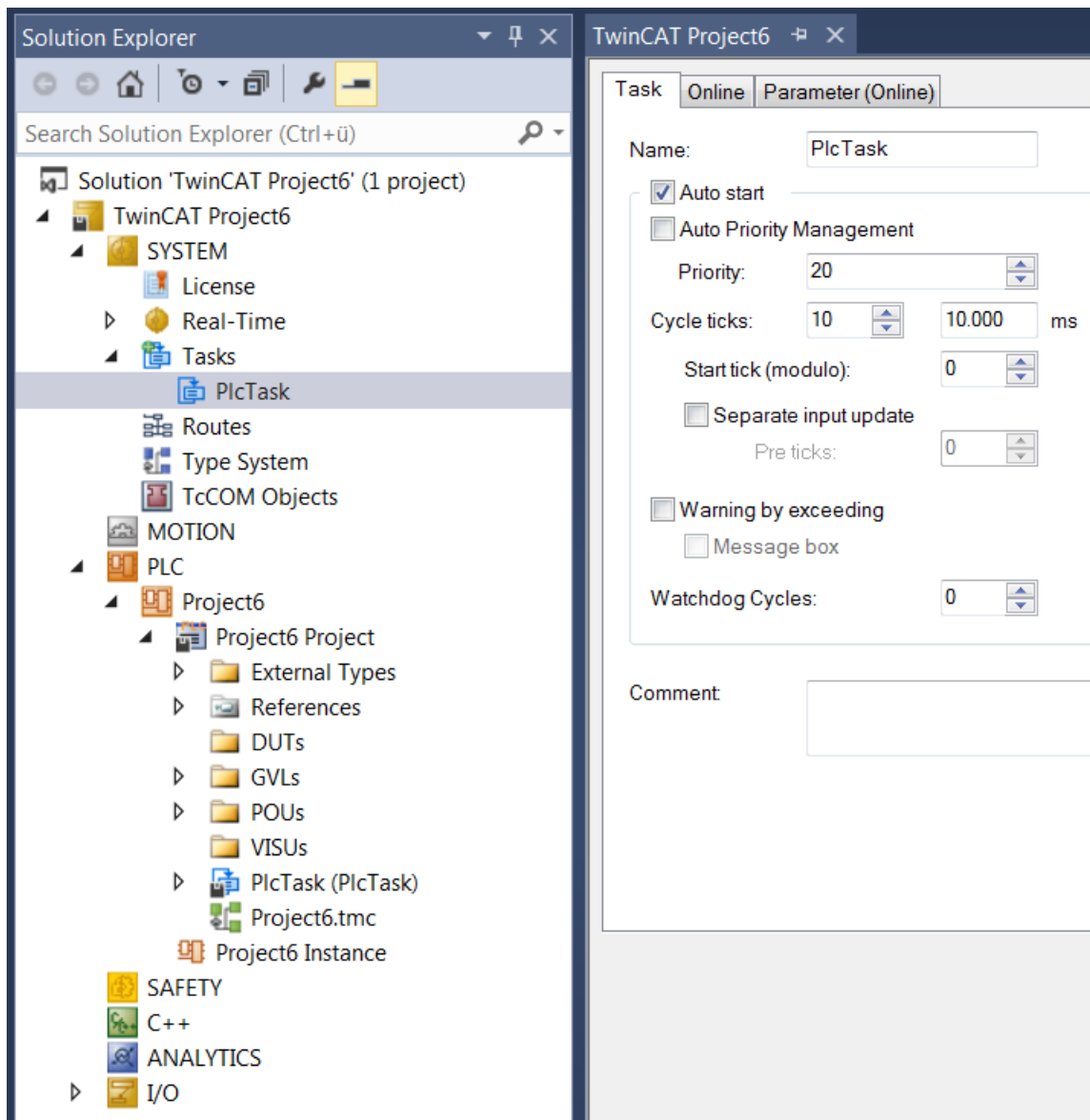
i For convenient operation and monitoring of the entire control program, a visualization can be used that displays the refrigerator and shows the operation of the simulation program. In TwinCAT, you can also program a visualization in the PLC area. When the project is started on the controller, the visualization starts without you having to make an entry. Depending on the programming, you can trigger opening and closing of the door by clicking an on/off switch, for example, or by adjusting the temperature preselection using the needle of a control knob. The process of creating a visualization is not described here.

Defining the programs to be executed in the task configuration

The preset task configuration contains the call for the MAIN program. For our sample project you have to add the call for the program "Simulation".

1. In the PLC project tree, drag the "Simulation" entry to the task reference (PlcTask).
 - ⇒ The "Simulation" program is added to the task configuration.
2. To view the task configuration, double-click the **PlcTask** entry in the PLC project tree.
 - ⇒ In the **Solution Explorer** the referenced task (**PlcTask**) is enabled under **SYSTEM > Tasks**.
3. Double-click the task to open the task's configuration in an editor.

⇒ In the PLC project tree at **PlcTask** you see the POU's that are called by the task: MAIN (entered by default) and Simulation. The editor of the **PlcTask** node in the SYSTEM area shows the corresponding cycle time for the referenced PlcTask. In this sample, the interval is 10 milliseconds. In online mode, the task will process the two function blocks once per cycle.



Defining the active PLC project

A TwinCAT controller can run multiple PLC projects. The first entry in the **Active PLC Project** drop-down list in the **TwinCAT PLC Toolbar Options** shows the currently active PLC project. If several PLC projects are present, you can select a PLC project via the drop-down list.

Checking the PLC project for errors

While entering code, TwinCAT immediately alerts you to syntax errors by means of a squiggly red line.

1. To get a syntax check over the whole PLC project, select the PLC project object "<PLC project name> Project".
2. Choose the command **Check all objects** from the context menu or from the **Build** menu.
 - ⇒ The results of the check are displayed in the **Error List** view.
3. If necessary, open the **Error List** view with the command **Error List** from the **View** menu.
4. You can then double-click the message to jump to the corresponding code position.

The command **Check all objects** can be used to check and compile all function blocks in the PLC node. An error is generated if the compilation reveals function blocks in the tree that are uncompileable, for example because they may only have been included for testing purposes. It is therefore advisable to run the command **Check all objects**, particularly for checking library function blocks.

The commands **Build** or **Rebuild** can be used to limit checking and compiling to function blocks that are actually used in the PLC project.

Further checks of the PLC project are performed when it is loaded onto the controller.

Only error-free PLC projects can be loaded onto the controller.

Compiling the PLC module



The commands **Build** or **Rebuild** can be used to compile your code used in the PLC project and check for syntactic accuracy.

Choose Target System



Now select the target device for your control program from the **Choose Target System** drop-down list in the **TwinCAT XAE Base Toolbar Options**:

- Select **<Local>** to load the control code directly into the local runtime of your programming device. (Select this options for the present sample.)
- If you want to select another target device, select **Choose Target System** from the drop-down list. Then select a preconfigured target device or browse the network for a target device, configure it and then select it.

Activation of the configuration


1. Click  in the **TwinCAT XAE Base Toolbar Options**.
⇒ A dialog appears asking whether you want to activate the configuration.
2. Click on **OK**.
⇒ A dialog appears asking whether TwinCAT should be restarted in Run mode.
3. Click on **OK**.
⇒ The configuration is activated, and TwinCAT is set to Run mode. In the taskbar shows the current status: . Activation also transfers the PLC project to the controller.


Loading the PLC project onto the PLC

- ✓ The PLC project was compiled without errors. See step [Checking the PLC program for errors \[► 34\]](#),
1. Select the **Login** command in the **PLC** menu or click on  in the **TwinCAT PLC Toolbar Options**.
⇒ A dialog box appears asking whether the application should be created and loaded.
 2. Click **Yes**.
⇒ The PLC project is loaded onto the controller. The engineering environment is now in online mode. The PLC modules are not yet in Run mode. In the **Solution Explorer** the following symbol appears before the PLC project object: . During the loading process, the **Error List** view displays information about the generated code size, the size of the global data and the resulting memory requirements on the controller.

Starting the program

If you have followed the entire tutorial up to this point, you can now use the PLC project on the PLC device.

1. Select the **Start** command in the **PLC** menu or click on  in the **TwinCAT PLC Toolbar Options ([F5])**.

⇒ The program is running. The PLC modules are in Run mode. In the **Solution Explorer** the following symbol appears before the PLC project object:  .

Monitoring and writing variable values once at runtime

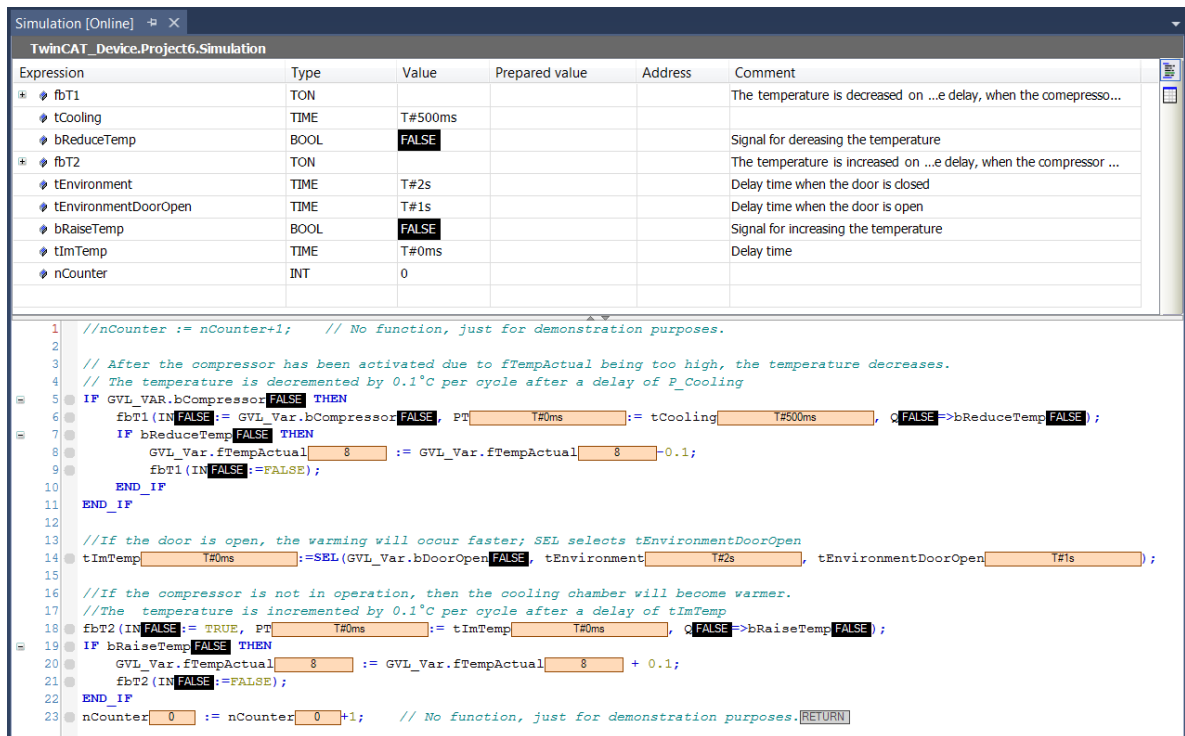
The following section illustrates the monitoring of the variable values in the various program blocks, and you are invited to set a certain variable value on the controller once from TwinCAT.

The actual values of the program variables can be seen in the online views of the function block editors or in watch lists. This sample is deliberately limited to monitoring in the function block editor.

✓ The PLC program runs on the controller.

1. Double-click the objects MAIN, "Signals", "Simulation" and "GVL_Var" in the PLC project tree to open the online views of the editors.

⇒ In the declaration part of each view, the table of expressions in the **Value** column shows the actual value of the variables on the controller.

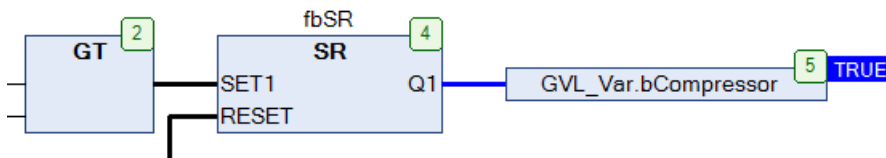


Expression	Type	Value	Prepared value	Address	Comment
* fbT1	TON				The temperature is decreased on ...e delay, when the comepresso...
tCooling	TIME	T#500ms			
bReduceTemp	BOOL	FALSE			Signal for decreasing the temperature
* fbT2	TON				The temperature is increased on ...e delay, when the compressor ...
tEnvironment	TIME	T#2s			Delay time when the door is closed
tEnvironmentDoorOpen	TIME	T#1s			Delay time when the door is open
bRaiseTemp	BOOL	FALSE			Signal for increasing the temperature
tImTemp	TIME	T#0ms			Delay time
nCounter	INT	0			

```

1 //nCounter := nCounter+1; // No function, just for demonstration purposes.
2
3 // After the compressor has been activated due to fTempActual being too high, the temperature decreases.
4 // The temperature is decremented by 0.1°C per cycle after a delay of P_Cooling
5 IF GVL_Var.bCompressor FALSE THEN
6   fbT1 (IN FALSE := GVL_Var.bCompressor FALSE, PT T#0ms := tCooling T#500ms, C FALSE => bReduceTemp FALSE);
7   IF bReduceTemp FALSE THEN
8     GVL_Var.fTempActual 8 := GVL_Var.fTempActual 8 -0.1;
9     fbT1 (IN FALSE :=FALSE);
10  END_IF
11 END_IF
12
13 //If the door is open, the warming will occur faster; SEL selects tEnvironmentDoorOpen
14 tImTemp T#0ms :=SEL(GVL_Var.bDoorOpen FALSE, tEnvironment T#2s, tEnvironmentDoorOpen T#1s);
15
16 //If the compressor is not in operation, then the cooling chamber will become warmer.
17 //The temperature is incremented by 0.1°C per cycle after a delay of tImTemp
18 fbT2 (IN FALSE := TRUE, PT T#0ms := tImTemp T#0ms, C FALSE => bRaiseTemp FALSE);
19 IF bRaiseTemp FALSE THEN
20   GVL_Var.fTempActual 8 := GVL_Var.fTempActual 8 + 0.1;
21   fbT2 (IN FALSE :=FALSE);
22 END_IF
23 nCounter 0 := nCounter 0 +1; // No function, just for demonstration purposes. RETURN
    
```


⇒ The monitoring in the implementation part depends on the implementation language: For non-Boolean variables, the value is always in a rectangular field to the right of the identifier. In the ST editor, this also applies to Boolean variables. This display is named "Inline Monitoring". In the graphical editors, the value of a Boolean variable is indicated by the color of the output link line: black for FALSE, blue for TRUE:



⇒ Note how the variable values in the different function blocks change. For example, the global variable list "GVL_Var" shows how the values of "fTempActual" and "bCompressor" change when the simulation program is executed.



One-time setting of variable values on the controller:

1. Set the focus to the online view of the global variable list "GVL_Var".
2. To specify a new setpoint, double-click the column **Prepared value** under the expression "fTempSet".
 - ⇒ An input field opens.


3. Enter the value "9" and exit the input field.
4. To set the door to open, click once in the **Prepared value** field of the "bDoorOpen" expression.
 - ⇒ The value TRUE is entered.
5. Click three more times to see that you can use this to set the prepared value to FALSE, then to empty again and then to TRUE again.
6. To write the prepared value TRUE once to the variable, select the command **Write values** in the **PLC** menu or click  in the **TwinCAT PLC Toolbar Options**.
 - ⇒ The two values are transferred to the **Value** column. The variable "bDoorOpen" no longer changes its value and the set temperature is now 9 degrees. The variable "tlmTemp" changes to the value 1 s, because the refrigerator door is now "open" and thus the heating by simulation should be faster than before (2 s).

Setting breakpoints and step-by-step execution at runtime

Debugging: For troubleshooting, you want to check the variable values at certain code positions. To do this, you can define breakpoints for processing and initiate step-by-step execution of the instructions.

- ✓ The PLC program is loaded onto the controller and is running.
1. Double-click the "Simulation" object to open the program in the editor.
 2. Place the cursor in the code line `nCounter:=nCoutner+1;` and press **[F9]**.
 - ⇒ The symbol  appears before the code line. It indicates that a breakpoint is set at this line. The symbol immediately changes to . The yellow arrow always points to the next instruction to be executed.
 3. Consider the value of the variable "nCounter" in the inline monitoring or in the declaration part of the **Simulation** program.
 - ⇒ The variable value no longer changes. The execution was stopped at the breakpoint.
 4. Press **[F5]** to restart the execution.
 - ⇒ The program stops again at the breakpoint after one cycle. "nCounter" was incremented by 1.
 5. Press **[F11]** to execute the next step
 - ⇒ RETURN at the end of the line `nCounter:=nCounter+1;` the instruction is highlighted in yellow
 6. Press **[F11]** again to execute the next step.
 - ⇒ The execution jumps to the editor of MAIN. Pressing **[F11]** repeatedly shows how the program is executed step-by-step. The instruction to be executed is again marked with a yellow arrow.
 7. To disable the breakpoint and return to normal execution, move the cursor into the code line again and press **[F9]**. Then press **[F5]** to restart the program execution.
 - ⇒ You can run through the program step-by-step and check variable values at certain code positions.

Executing a single cycle at runtime

- ✓ The PLC program is loaded onto the controller and is running.
1. Observe the line `nCounter:=nCounter+1;` again in the **Simulation** program.
 2. Click  in the **TwinCAT PLC Toolbar Options** to execute a single cycle.
 - ⇒ The execution runs through a cycle and stops again at the breakpoint. "nCounter" was incremented by 1.
 3. Click the button three more times to see the individual cycles. Then press **[F5]** again.
 - ⇒ The program runs again without stop and without forced values. The variable "tlmTemp" has the value 1 s again.

3 Tips and tricks

On the following pages you will find some useful information that may be helpful to you in the planning of a TwinCAT 3 project. These tips and tricks include the following points:

- Recommended steps before [Project delivery](#) [[▶ 38](#)]
- [New properties and features](#) [[▶ 42](#)] of TwinCAT 3 versions

3.1 Project delivery

Once the development of a PLC project is completed and before the project is delivered, we recommend checking the following steps and, if they make sense in relation to the project, to perform them.

Command/step	Where to find it	Purpose	Further information
<p>Activating the "Pin Version" option</p>	<p>TwinCAT 3 project > Double-click the SYSTEM node > "General" tab</p>	<p>On opening the project, the TwinCAT 3 Engineering version defined in the project is automatically used if it exists on the engineering computer.</p> <p>For the avoidance of source code changes - when activating/logging in later - that could be triggered by the use of a different engineering version</p>	<p>TE1000 XAE\ Technologies\ Remote Manager\ TwinCAT integration\ Pin Version project setting</p>
<p>Setting the compiler version of the PLC application project to the "newest"</p>	<p>PLC project > Right-click the PLC project object > Select the "Properties" command > Open the "Compile" category > Set the "Compiler version" setting to "newest"</p>	<p>The compiler version used for application projects is then automatically the one that matches the engineering version that is loaded in the Remote Manager (see point 1: Pin Version). This is the intended method of use of the compiler version.</p>	<p>Category Compile [► 910]</p>

Command/step	Where to find it	Purpose	Further information
<p>Setting the PLC library references to a fixed version (no use of "always newest"/"*")</p>	<p>PLC project</p> <p>For all library/placeholder references simultaneously with just one command:</p> <ul style="list-style-type: none"> > Right-click the References node > Select the command "Use effective version" <p>or for individual or several libraries/placeholders (by multiple selection):</p> <ul style="list-style-type: none"> > Right-click a library reference underneath the References node > Select the command "Use effective version" <p>or for individual libraries/placeholders:</p> <ul style="list-style-type: none"> > Mark the library reference underneath the References node > Select a fixed version in the Properties window <p>or for individual placeholders:</p> <ul style="list-style-type: none"> > Right-click the References node > Open the placeholder dialog > Select a fixed version in the "Library" column 	<p>For the avoidance of an Online Change query - when logging in later - that would be triggered by the automatic use of a newer library version if a library reference to "always newest"/"*" is triggered and a newer library version is available in the library repository for this library</p>	<p><u>Recommendations and notes [▶ 268]</u></p>

Command/step	Where to find it	Purpose	Further information
<p>Checking the settings of the target archive</p> <p>Note: If you should change the settings, it is necessary to reactivate the configuration or the boot project in order for these settings to be accepted.</p>	<p>PLC project</p> <ul style="list-style-type: none"> > Double-click the PLC project > "Settings" tab 	<p>For the avoidance of the inadvertent passing-on of the project sources or libraries due to saving on the target system</p> <ul style="list-style-type: none"> • Potential advantages if the project sources are located on the target: <ul style="list-style-type: none"> ◦ The PLC project opened in the XAE can be compared in detail with the version on the target ("Compare Project with Target" command). ◦ The PLC project can be loaded and opened from the target ("Open Project from Target" command). • Potential disadvantages if the project sources are located on the target: Those with access to the target system can read/copy the sources. 	<p>Settings tab 926</p>

For standard machines, the use/activation of the following options is usually helpful. In addition, the two options are necessary if a complete machine update is to be carried out at file level.

Option	Where to find it	Purpose	Further information
Option "Use relative NetIds" in the route settings	TwinCAT 3 project > Expand the SYSTEM node > Double-click the Route node > "NetId Management" tab	If the option "Use relative NetIds" is activated, then the first four digits correspond to the project NetId of a placeholder and are set according to the actual NetId of the target system. This allows a project or a configuration to be used on another, identical target system without manual adaptation of the project NetId. Example: <ul style="list-style-type: none"> • Target system with the AMS NetId 172.17.35.70.1.1 • NetId of the first device (e.g. EtherCAT Master) <ul style="list-style-type: none"> ◦ if the option "Use relative NetIds" is not enabled: 172.17.35.70.2.1 ◦ if the option "Use relative NetIds" is enabled: [172.17.35.70].2.1 	TE1000 XAE\ System\ System node\ System sub-node\ Routes and TE1000 XAE\ System\ Machine update at file level\ Execute complete machine update
Option "Virtual device names" in the adapter settings of all network and USB devices	TwinCAT 3 project > Expand the I/O node > Expand the Device node > Double-click the device, e.g. the EtherCAT Master > "Adapter" tab	If the option "Virtual device names" is activated, then the spoken name or display name is used as a reference to the device. The system then uses the device name, not the MAC address.	TE1000 XAE\ I/O\ EtherCAT\ EtherCAT Master\ Adapter and TE1000 XAE\ System\ Machine update at file level\ Execute complete machine update

3.2 New properties and features

3.2.1 TwinCAT 3.1 Build 4024

The following new properties and features are available with TwinCAT 3.1 Build 4024.

Category	Feature	Further information	Available from build revision
Development support	"Go to definition" command available in the PLC process image	Command Go To Definition [▶ 880]	4024.0
	Menu command and hotkey for uncommenting or canceling an uncommenting	Command Comment Selection [▶ 882] Command Uncomment Selection [▶ 882]	4024.0
	Identification of unlinked allocated variables (AT%I, AT%Q)	see below	4024.10
Storage format	Optional storage format Base64	Command Properties (object) [▶ 901]	4024.0
Compiler	Conditional compilation – in addition to the implementation editor – now also available in the declaration editor	Conditional pragmas [▶ 837]	4024.0
Object-orientated programming	Extended monitoring of an interface variable: Symbol path and online data of the currently assigned FB instance are displayed under the interface variables in the monitoring area (declaration editor, monitoring list).	Object Interface [▶ 102]	4024.0
	Mini-icons show access modifier: Mini-icons on the object symbols in the project tree show whether the object is private, public or protected.	Object Method [▶ 90]	4024.0
	ABSTRACT keyword	ABSTRACT concept [▶ 201]	4024.0

Category	Feature	Further information	Available from build revision
Online features	{attribute 'to_string'} for enums: This attribute is used in order to textually query the name of an enum element by means of TO_STRING/ TO_WSTRING.	Attribute 'to_string' [▶ 836]	4024.0
	Exception-Handling via __TRY/ __CATCH (for 32-bit runtime systems)	__TRY, __CATCH, __FINALLY, __ENDTRY [▶ 737]	4024.0
	Memory reserve for Online Change	Configuring the memory reserve for Online Change [▶ 133]	4024.0
	Details button in the message window when logging-in a changed PLC project	Updating the PLC project on the PLC [▶ 250]	4024.0
	Command "Go to instance"	Command Go to Instance [▶ 880]	4024.0
	Error analysis with core dump	Error analysis with core dump [▶ 247]	4024.11
Miscellaneous	Option "Generate tpy file"	Category Compile [▶ 910]	4024.0

Identification of unlinked allocated variables (AT%I, AT%Q)

With the help of the summary described below, you can obtain an overview of the allocated variables (AT%I, AT%Q) that are not linked. As a result of this you can, for example, obtain an overview of the links or the project status.

Procedure:

- Double-click the input or output process image of a PLC task in the Solution Explorer (e.g. double-click the object "PlcTask Inputs" or "PlcTask Outputs").
 - ⇒ A summary opens showing the allocated inputs or outputs of this PLC process image (see below, first screenshot).
- Click the header of the second table column (column header: "[X]").
 - ⇒ Only those allocated inputs or outputs that are not linked are still shown (column header changes to : "[]" (see below, second screenshot).
- Click the header of the second table column again.
 - ⇒ All allocated inputs or outputs are shown again (column header changes to "[X]").

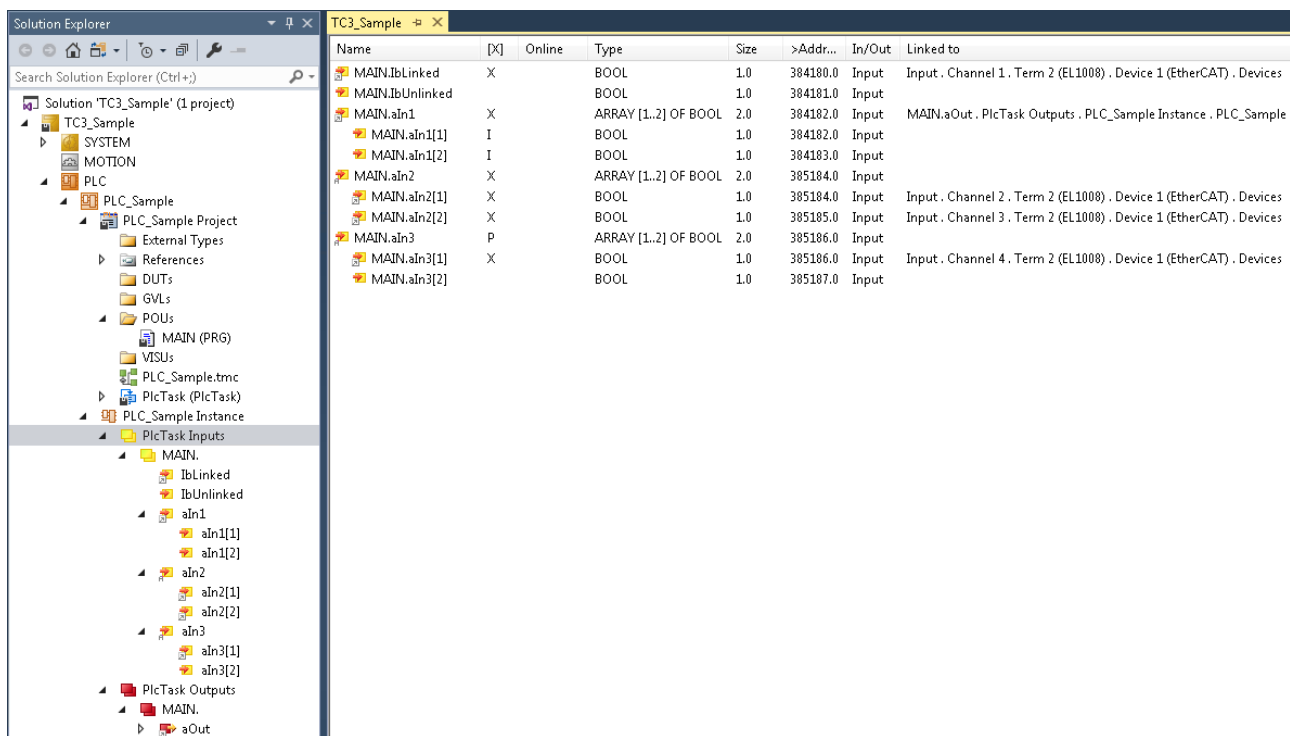
Notes:

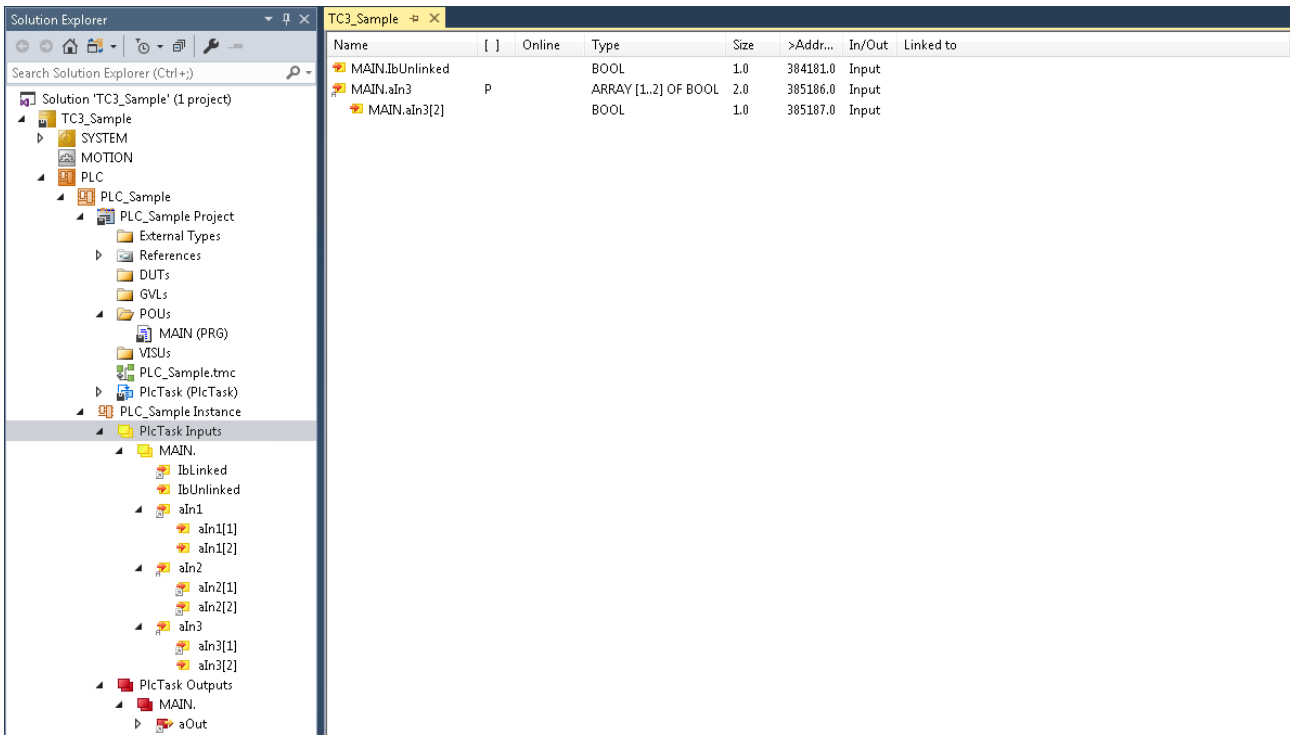
- With the help of this summary you can create new links or change existing links. To do this, right-click a variable/line and select the command **Change link**.
- To display the sub-elements of a structured variable or an array, click the **Show sub-elements** button inside the **TwinCAT XAE Base** toolbar.

Values of the column "[X]" or "[]":

Column value	Meaning	Example (see *)	Example variable in the screenshots (see below)
Empty	Not linked		IbUnlinked
X	Linked	Individual linked variable	IbLinked MAIN.aIn3[1]
		Array where the array itself is linked	MAIN.aIn1
		Array where all elements of the array are linked separately	MAIN.aIn2
P	Partially linked	Array where at least one, but not all elements of the array are linked separately	MAIN.aIn3
I	Indirectly linked (via a parent variable)	Elements of an array where the array itself is linked so that the array elements are indirectly linked as well	MAIN.aIn1[1]

* The named array examples are also transferrable to structured variables (e.g. structure instance) and their sub-elements.





3.2.2 TwinCAT 3.1 Build 4026

The following new properties and features are available with TwinCAT 3.1 Build 4026.

Category	Feature	Further information	Available from Build Revision
Development support	Cross Reference List: New "Context" column	Command Cross Reference List [▶ 940]	4026.0
	List components: Extension of the function with categories and highlighting of the entered characters.	Using the input wizard [▶ 135]	4026.0
	Smart tag function: Simple declaration of variables from the implementation part.	Using the input wizard [▶ 135]	4026.0
	PLC Bookmarks	Set and use bookmarks [▶ 158]	4026.0
ST editor	Dark Theme (offline)	Dialog Options - Text editor [▶ 985]	4026.0
	Highlighting of all places of use of the variable on which the cursor is placed.	ST Editor [▶ 624]	4026.0
	Incremental search	ST Editor [▶ 624]	4026.0
	Rectangle selection	ST Editor [▶ 624]	4026.0
CFC editor	Setting for execution order Default: Automatic data flow mode	Command Properties (object) [▶ 905]	4026.0
	Temporary display of the execution order	Command Display Execution Order [▶ 1014]	4026.0
	Easier insertion of multiple elements from the toolbox	CFC Editor [▶ 666]	4026.0
	Drag and drop of variables and objects from the declaration section and from the Solution Explorer	CFC Editor [▶ 666]	4026.0
	Changing the pin arrangement via drag and drop	CFC Editor [▶ 666]	4026.0
Object-orientated programming	For input parameters with initial value, no variables have to be passed in a call.	Method call [▶ 199]	4026.0
	IntelliSense for additional FB_init parameters	FB_init [▶ 850]	4026.0
Online features	Structured representation of inherited variables	Command Presentation of inheritance - Simple, Structured [▶ 964]	4026.0
	"Go to implementation" and "Go to reference" commands	Command Go to implementation [▶ 880] Command Go to reference [▶ 880]	4026.0
	Filter option in the "Select online status" dialog	Monitoring in Programming Objects [▶ 222]	4026.0

Category	Feature	Further information	Available from Build Revision
Libraries	Use of a PLC project as referenced library.	Use PLC project as referenced library	4026.0
	New options for libraries in the PLC project properties: "Allow implicit checks for compiled libraries" and "Force Qualified_only for library access".	Category Common [► 906]	4026.0
	Improve navigation in the library manager through linking.	Library Manager [► 276]	4026.0
	Ability to activate and deactivate repositories in the library manager.	Library Repository [► 272]	4026.0
	Distinguish between the commands for adding placeholders and libraries.	Add library without placeholder resolution command [► 359]	4026.0
	Multiple selection in the "Add Library" dialog	Command Add library [► 358]	4026.0
	Export of the values of a parameter list as .csv file	Object Parameter List [► 73]	4026.0
Operators and variables	Determine the name of local objects with the <code>__POUNAME</code> operator and the line number with the <code>__POSITION</code> operator.	__POUNAME [► 743] __POSITION [► 743]	4026.0
	Generic constants	Generic constant variables - VAR GENERIC CONSTANT [► 689]	4026.0
Miscellaneous	Representation of the type system: Library Tc3_Global_Types ExternalTypes.tmc instead of the External Types folder	Using global data types [► 61]	4026.0
	Save PLC projects as reusable project templates.	Command Save as PLC project template	4026.0

3.3 Further helpful properties and features

In addition to the properties and features that are made available with a certain TC3.1 version, the engineering environment contains the following helpful properties and features, among others.

Category	Feature	Further information
Development support	Hotkey: For many functions there are hotkeys that can be individually adapted if necessary.	Hotkey [► 51]
	Option "Activate folder view": For an optional folder view in the PLC process image and in the link dialog	see below
	Refactoring: For the convenient subsequent changing of a program	Refactoring [► 160]
	Object-oriented programming options: Can be used for the creation of easily readable and extendable software	Object-oriented programming [► 169]
Orientation and navigation	Find locations where the cross reference list is used	Using the Cross Reference List to find Occurrences [► 157]
	Find declaration	Finding Declarations [► 157]
Online features	Implicit monitoring functions: e.g. to check the limits of arrays during the runtime	Using function blocks for implicit checks [► 163]
	Flow Control: For tracking the program execution	Flow Control [► 219]
	Monitoring lists: For the clear monitoring of variable values	Using Watchlists [► 226]
	Breakpoints and step-by-step execution of the program: For finding errors in the program	Use of breakpoints [► 211] Stepwise processing of the program (stepping) [► 213]
Libraries	Using libraries: For the creation and use of collections of reusable objects	Using libraries [► 266]
	Library placeholders: For the convenient selection of a certain library version for a placeholder that is directly or indirectly referenced in the PLC project	Library placeholders [► 279]
	Command "Use effective version": For defining a library reference to a certain library version	Command Set to Effective Version [► 364] Project delivery [► 38]

Optional folder view in the PLC process image and in the link dialog

Option: Checkbox **Activate folder view**

- Activated: The variables in the PLC process image and in the PLC area of the link dialog are displayed as expandable folder levels.
- Deactivated: The variables in the PLC process image and in the PLC area of the link dialog are displayed as flat lists.

Dialog: Double-click the PLC node in the **Solution Explorer > PLC settings** tab

Sample:

Function block FB_Sample

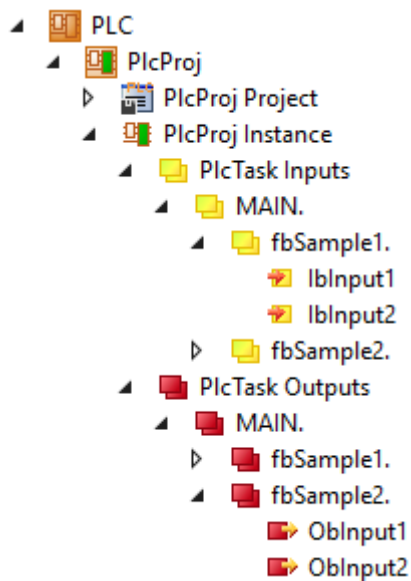
```
FUNCTION_BLOCK FB_Sample
VAR
  IbInput1  AT%I*  : BOOL;
  IbInput2  AT%I*  : BOOL;
  ObInput1  AT%Q*  : BOOL;
  ObInput2  AT%Q*  : BOOL;
END_VAR
```

MAIN program

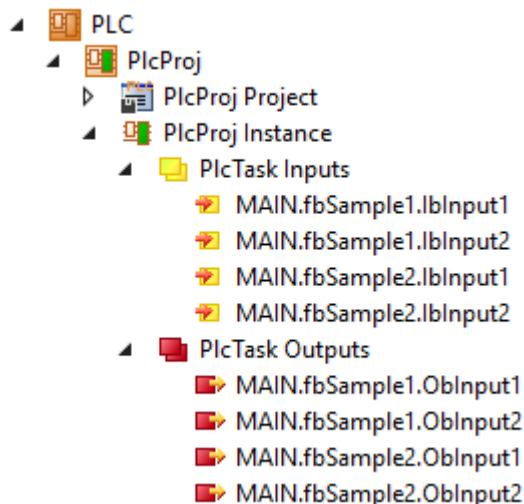
```
PROGRAM MAIN
VAR
  fbSample1      : FB_Sample;
  fbSample2      : FB_Sample;
END_VAR
```

PLC process image:

- Case 1: **Activate folder view** checkbox is activated.

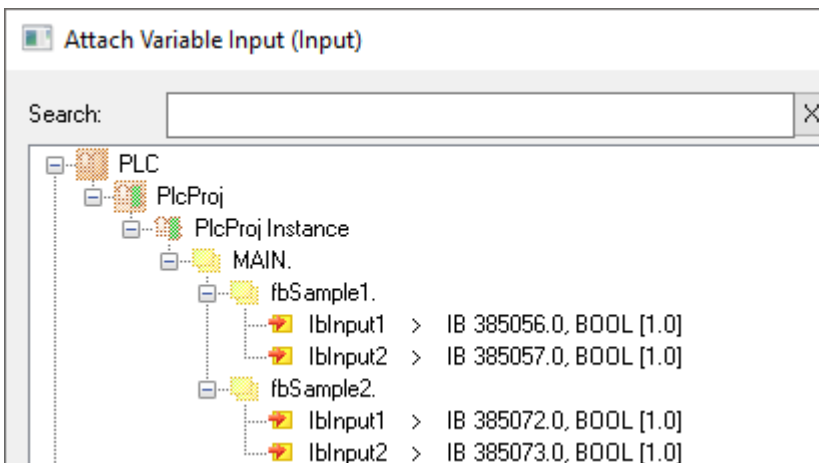


- Case 2: **Activate folder view** checkbox is deactivated.

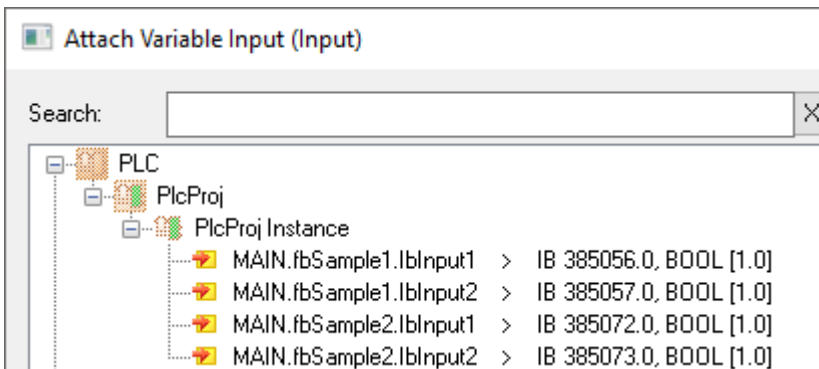


Link dialog:

- Case 1: **Activate folder view** checkbox is activated



- Case 2: **Activate folder view** checkbox is deactivated



3.3.1 Hotkey

Some commands can be executed by hotkey in the TwinCAT 3 development environment. Below you will find a summary of these commands with their default hotkeys. If necessary, you can adapt and/or extend the configuration of the hotkeys.

Category	Command	Default hotkey	Further information
Standard commands	Undo	Ctrl+Z	Standard Commands [▶ 873]
	Redo	Ctrl+Y	
	Copy	Ctrl+C	
	Cut	Ctrl+X	
	Paste	Ctrl+V	
	Select All	Ctrl+A	Command Select All [▶ 873]
	Delete	Delete	Command Delete [▶ 873]
Development support	Go To Definition	F12	Command Go To Definition [▶ 880]
	Search for references	Shift+F12	Command Find all references [▶ 881]
	Input Assistant	F2	Command Input Assistant [▶ 873]
	Change focus between declaration and implementation editor	F6	
Comment	Uncomment	[Ctrl+K] + [Ctrl+C]	Command Comment Selection [▶ 882]
	Cancel uncomment	[Ctrl+K] + [Ctrl+U]	Command Uncomment Selection [▶ 882]
Online commands	Start	F5	Command Start [▶ 958]
	Stop	Shift+F5	Command Stop [▶ 959]

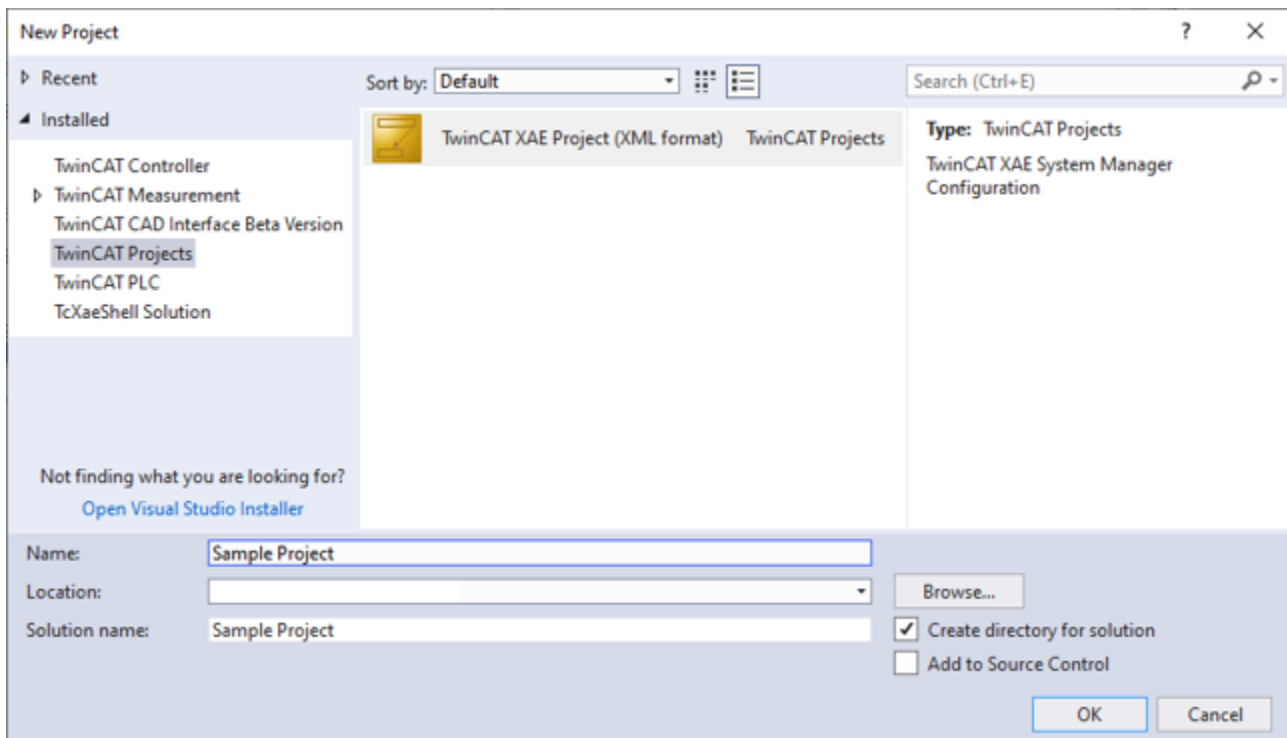
3.4 Renaming a project

Scenario

An existing project is to be reused for a new machine. For this purpose, the project is to be given the name of the new machine.

Challenge

When creating a project, you will be asked whether a Solution directory should be created. This option is the default behavior. If it is selected, the TwinCAT 3 project will be created in this directory.



The Solution file (*.sln) for the project contains a reference to all subprojects. If a TwinCAT 3 project is renamed, only the project file is renamed, not the folder. Thus, after renaming the project, the old folder name of the TwinCAT 3 XAE project will still exist on the hard disk, even though it is not located in the solution.

Solution options

- Select generic folder names that have no reference to a project.
- Disable the option **Create directory for solution**, then the solution file is stored next to the project file of the TwinCAT 3 XAE project.
- If the Solution file only contains one TwinCAT 3 project, then the TwinCAT 3 project can be copied manually to another folder without the *.sln file and the folder in which the TwinCAT 3 project is saved. Double-clicking the *.tsproj file also opens the TwinCAT 3 XAE Shell. You can rename the project accordingly in the XAE Shell. When saving the project, you will automatically be asked whether a new Solution file should be created.
- Copy the entire directory and adapt the folder names of the subprojects manually. Subsequently, these folder names must also be manually swapped in the *.sln file.

4 Creating and configuring a PLC project

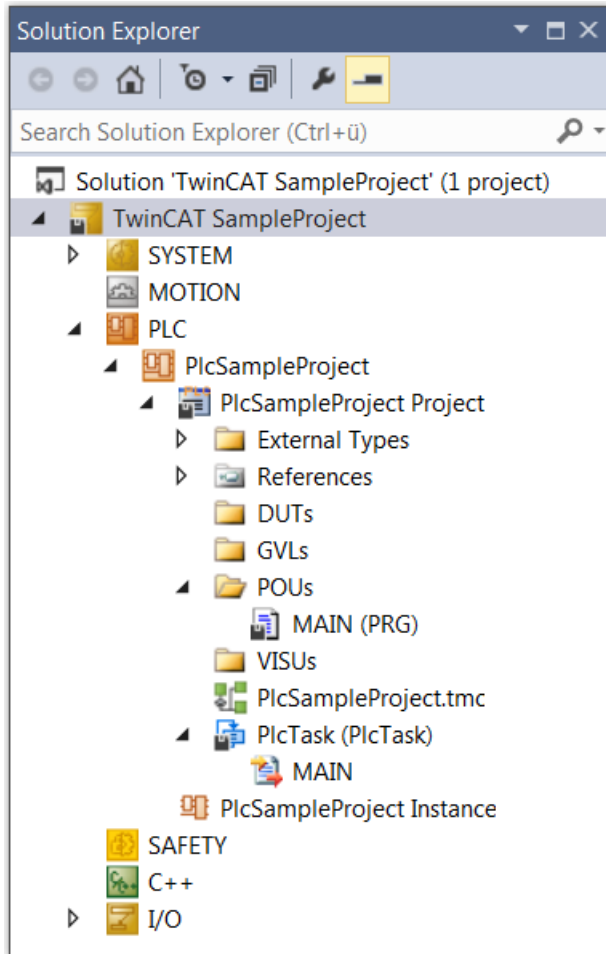
What is a PLC project?

- A project contains the objects required to create a controller program:
 - Pure programming blocks, for example programs, function blocks, functions, GVLs.
 - Objects that are additionally required in order to be able to run the program on a PLC. For example, referenced tasks, library managers and visualizations.
- In a TwinCAT project you can program several PLC projects and run them on a target device.
- TwinCAT manages project-specific function blocks in the view Solution Explorer under the PLC nodes.
- There are templates containing certain objects for the creation of projects.
- Basic configurations and information on the project are defined in the project settings and project information. For example:
 - Compiler settings
 - Author
- You save a project as a file in the file system. Optionally, you can package it together with project-relevant files and information in a project archive. Storage in a source code management system such as Microsoft Team Foundation Server (TFS) or SVN is also possible.
- Each project contains the information regarding the TwinCAT version with which it was created. If you open it in a different version, TwinCAT will inform you of possible or necessary updates.
- You can compare projects and export and import individual objects.
- You can protect a project against being changed and even completely against being read. Through the use of user administration you can purposefully control access to the project and even to individual objects in the project.

4.1 Creating a standard project

1. Select the command **New > Project** in the menu **File**.
 - ⇒ The dialog **New Project** opens.
2. Select the template **TwinCAT Projects > TwinCAT XAE Project** and enter a name, for example here "Sample Project" and a storage location in the file system.
3. Quit the dialog with **OK**.
 - ⇒ A new solution opens in the **Solution Explorer**. The project name "TwinCAT Sample Project" appears in the title bar of the main window.
4. Mark the PLC object in the project tree and select the command **Add New Item...** in the menu **Project** or in the context menu.
 - ⇒ The dialog **Add New Item <TwinCAT project name>** opens.
5. Select the **Standard PLC Project** in the category **Plc Templates** and enter a name (here once again "Sample Project").
6. Quit the dialog with **Add**.

⇒ The following structure is created in the view **Solution Explorer**.



⇒ With the selected template the following basic objects appear automatically under the PLC project object (**PlcSampleProject**): a PLC project (**PlcSampleProject project**) and a project instance (**PlcSampleProject instance**). The PLC project contains a library manager (**References**), the standard program block **MAIN** and a task reference (**PlcTask**). The tasks referenced there (**PlcTask**) defines the execution of the program block MAIN. In addition, the structure folders **External Types**, **DUTs**, **GVLs**, **POUs** and **VISUs** appear automatically.

The library manager already contains the standard libraries with basic blocks such as counters, timers and string functions that can be used subsequently for programming.

If you now fill MAIN with error-free code, you can load it to the controller and run it without the need for further programming objects.

See also:

- [Programming a PLC project \[▶ 66\]](#)
- [Using libraries \[▶ 266\]](#)

4.2 Adding objects

The following instructions indicate some possibilities to create objects in the PLC project.

✓ A PLC project is open.

1. Select an entry in the PLC project tree, for example the folder **POUs**.
2. Select the command **Add** in the context menu.

⇒ Depending on the entry selected in the tree, TwinCAT offers suitable objects for selection. The following objects are available to you:

- [Object POU \[▶ 78\]](#)
 - [Object Method \[▶ 90\]](#)

- [Object Property](#) [▶ 96]
 - [Object Action](#) [▶ 87]
 - [Object Transition](#) [▶ 88]
 - [Object POUs for implicit checks](#) [▶ 163]
 - [Object DUT](#) [▶ 75]
 - [Object Global Variable List](#) [▶ 72]
 - [Object Referenced Task](#) [▶ 131]
 - [Object Recipe Manager](#) [▶ 228]
 - [Object Recipe Definition](#) [▶ 232]
 - [Object Image Pool](#) [▶ 146]
 - [Object Interface](#) [▶ 102]
 - [Object Parameter List](#) [▶ 73]
 - [Object "Text list"](#) [▶ 144]
 - [Object "Class Diagram](#) [▶ 133]"
 - [Object "Visualization" Manager](#) [▶ 387]
 - [Object "Visualization"](#) [▶ 402]
3. For example, select the object **POU** and in the dialog **Add POU**, which then appears, select the type **Program** with implementation language "Structured Text (ST)" and the name "Prog". Click on **Open**.
 - ⇒ TwinCAT adds a program object **Prog** in the PLC project tree below the selected entry.
 4. Select an object in the tree and select the command **Properties** (object) in the context menu.
 - ⇒ The **Properties** view with object-relevant categories becomes active. If you use a user administration you could, for example, restrict access to the object here.
 5. Select an entry in the tree below which you wish to create a folder in order to collect objects in it.
 6. Select the command **Add > New Folder** in the context menu.
 - ⇒ The folder appears in the PLC project tree.
 7. Give the folder a name and confirm it.
 8. Select an object in the PLC project tree and move it to a different position, for example into the folder, by dragging it with the mouse inside the project tree.

See also:

- TC3 User Interface documentation: Command Add Existing Item (Object)
- TC3 User Interface documentation: [Command Properties \(object\)](#) [▶ 901]
- TC3 User Interface documentation: Command New folder

4.3 Changing the compiler version

The version of the compiler with which the current project code for use on the target device is generated is defined in the project properties.

The compiler version is independent of the TwinCAT version. Therefore constant program code will be generated from the source code if the same compiler version is set, even if this takes place from different TwinCAT versions.



If you open a project in which the latest compiler version is not set, the dialog **Project Environment** appears with the corresponding information and the option to update directly.

✓ A PLC project is open.

1. Mark the PLC project and select the command **Properties** in the menu **Project** or the context menu.

- ⇒ The PLC project properties open in the editor window.
- 2. Select the category **Compile**.
- 3. In the group field **Solution options**, select the desired fixed version.
- ⇒ The change is effective immediately.

See also:

- TC3 User Interface documentation: Command Properties (project) > [Category Compile](#) [► 910]

4.4 Open a TwinCAT 3 PLC project

- ✓ TwinCAT XAE is started. A TwinCAT project is open.
- 1. Mark the PLC object in the view **Solution Explorer** and select the command **Add Existing Item** in the menu **Project** or the context menu.
- 2. In the dialog **Open**, select the desired TwinCAT 3 PLC project or project archive or a library from the file system. To search, you can set the file filter in the bottom right-hand corner of the dialog.
- 3. Click on **Open**.
- ⇒ The following cases are possible:
 - You have selected a project that was saved with a newer TwinCAT version:
Such a project may contain data that cannot be loaded. You can still open the project, but you need to consider the following: The project may behave in an unexpected manner because it cannot be completely interpreted. Objects that TwinCAT could not load completely or at all are marked in the view **Solution Explorer** in red and with the text "[unknown]" or "[incomplete]". TwinCAT cannot display objects in the editor that are unknown in the current version. TwinCAT displays incomplete objects in the editor with a warning that the displayed content may not correspond to the original. TwinCAT cannot save incomplete projects under their original name. This is indicated by a write-protection note in the top right-hand corner. You can save the file under a different name.
 - You have selected a project after whose last change TwinCAT was not terminated properly, but the project option **Auto Save** was activated:
The dialog **Auto Save Backup** for handling the backup copy now opens.

See also:

- [Open a TwinCAT 2 PLC project](#) [► 57]
- TC3 User Interface documentation: [Command Project/Solution \(Open Project/Solution\)](#) [► 862]
- TC3 User Interface documentation: [Command Add Existing Item \(Project\)](#) [► 863]

4.5 Open a TwinCAT 2 PLC project

- ✓ TwinCAT XAE is started. A TwinCAT project is open. You should be aware of the restrictions described below the following guide.
- 1. Mark the PLC object in the **Solution Explorer** view and select the **Add Existing Item** command in the **Project** menu or the context menu.
- 2. In the **Open** dialog, select the desired Plc 2.x project or library from the file system. To search, you can set the file filter in the bottom right-hand corner of the dialog.
 - ⇒ After that the TwinCAT 2.x converter starts automatically.
- 3. The TwinCAT 2.x converter checks whether the project can be compiled without error. If so, it processes the project automatically.
If the project contains visualization objects with placeholder variables that the converter cannot resolve, the respective visualizations are integrated directly as a grouping in place of the visualization references.
- 4. Library conversion: If a library for which no conversion rule has been defined yet is referenced in the project to be opened, the dialog **Converting a library reference** appears. Define here whether and how the converter should replace the previous library reference by an up-to-date one. If in doing so you select a library for which the project information is missing, the **Project information** dialog appears, which you have to fill out.
- ⇒ The converter loads the adapted project.

Restrictions when reusing a TwinCAT 2.x project in TwinCAT 3.1

Automatic syntax adjustments only with previous compilability

I The syntax differs between TwinCAT 2 (TC2) and TwinCAT 3 (TC3) in certain respects (e.g. when initializing arrays). Please note that the converter only adapts the syntax at these code positions if the TC2 project to be converted can be compiled on the TC2 side.

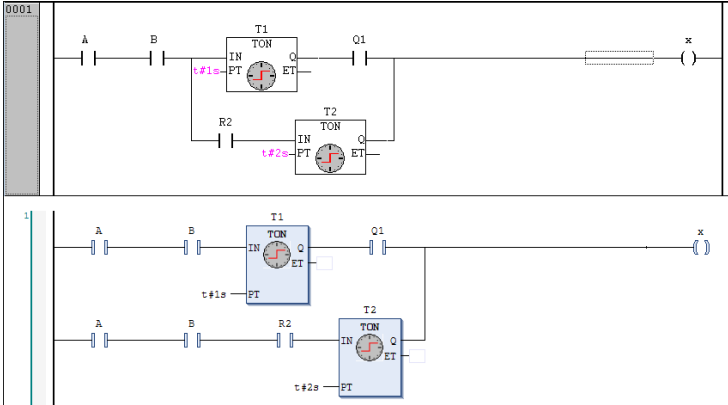
Specifically, this means:

During the conversion process, the converter contained in TC3 first compiles the selected TC2 project with a TC2 compiler. Syntax adjustments of the TC2 code are only made during the creation of the TC3 SPS project if this TC2-side compilation is successful. The prerequisite for a successful TC2-side compilation is that all necessary TC2 libraries are made available to the converter. This can be achieved, for example, by inserting the libraries referenced in the TC2 project into the folder C:\TwinCAT\3.1\Components\Plc\Converter\Lib, or by notifying the converter of the locations of the TC2 libraries via the displayed conversion dialogs.

Compilation	<p>It must be possible to compile the project in TwinCAT 2.x PLC Control without compilation errors. Nevertheless, TwinCAT issues warnings during the compilation. These are evoked by implicit conversions that can lead to the loss of information (for example due to the change of a sign).</p> <p>TwinCAT 3.1 tests case statements against the switch variable: CASE USINT OF INT is not checked in TwinCAT 2.x, but an error message is displayed when importing into TwinCAT 3.1.</p>
Libraries	<p>All variables and constants used in a library also have to be declared in this library. It must be possible to compile the library in TwinCAT 2.x without errors.</p>
Syntactic and semantic restrictions	<ul style="list-style-type: none"> • FUNCTIONBLOCK is no longer a valid keyword in place of FUNCTION_BLOCK • A ":" must follow after TYPE (declaration of a structure). • ARRAY initialization must be enclosed in parentheses. • A local declaration of an enumeration is no longer possible, except within TYPE: in the case of a TwinCAT 2.x import a local enumeration declaration is automatically converted into an explicit type definition. • INI is no longer supported (you have to replace this in the code by the Init method). • In function calls, it is no longer possible to mix explicit parameter assignments with implicit assignments. Therefore the order of the parameter input assignments can be changed: <pre>fun(formal1 := actual1, actual2); // → Fehlermeldung fun(formal2 := actual2, formal1 := actual1); // gleiche Semantik wie folgende Zeile: fun(formal1 := actual1, formal2 := actual2);</pre> • TwinCAT 2.x pragmas are not converted. They generate a warning in TwinCAT 3.1. • The TRUNC operator now converts to the data type DINT instead of INT; in the case of a TwinCAT 2.x import, TwinCAT 3.1 automatically adds an appropriate type conversion.
Memory	<ul style="list-style-type: none"> • In contrast to TwinCAT 2, TwinCAT 3.1 has an 8-byte alignment. For further details and samples see Reference Programming > Alignment [► 791]. • Instances of the data type STRING are pre-initialized in TwinCAT 2 by zeroing the entire memory area. In TwinCAT 3.1, however, only the first byte of such an instance is zeroed. Due to the zero termination, the instance is also empty.

Visualization

Placeholders and their replacement	Placeholder	VAR_INPUT	Use	Replacement
	MAIN. \$LocalVar\$.aArr[0]	localVar: MyStruct;	localVar.aArr[0]	localVar := MAIN.myStructVar
	\$Var\$.aArr[0]	Var : MyStruct;	Var.aArr[0]	Var := MAIN.myStructVar
	MAIN.myStructVar. aArr[\$Index\$]	Index : INT;	MAIN.myStructVar. aArr[Index]	Index := 0
Problematic placeholders	<ul style="list-style-type: none"> Placeholders inside a text: Text: \$axle\$-Axis Correction: localVar : STRING; Text: %s-Axis TextVariable : localVar Placeholder describes only part of a variable name: axis\$axis\$spur\$spur\$.fActPosition Correction: Define only one placeholder for the placeholder axis\$axis\$spur\$spur\$. axis_spur : MyFunctionBlock; Then transfer the corresponding instance of the function block directly. axis_spur := MAIN.axis1spur2; Placeholder is replaced by an expression: \$Expression\$ → MAIN.var1 + MAIN.var2 Correction: You must transfer the expression to a help variable and then transfer this help variable as an instance. Placeholder describes a program name: \$Program\$.bToggle → MAIN.bToggle D The converter cannot transfer this form of placeholder to TwinCAT 3.1. However, they are rarely used in practice. Placeholder is replaced by various types: \$Var\$ → Replacement 1: MAIN.n (INT) → Replacement 2: MAIN.st (STRING) Correction: Define two different placeholders for this in the interface. The visualization is located in a library. You replace the placeholder later from any project where you use the visualization. Correction: You must replace the TYPE_NONE data types manually here. However, there is a possibility for you to integrate the library in a project and for the placeholder to be replaced properly. If you now import this project, the data type will also be determined correctly in the library. 			
Non-importable elements	Trend, ActiveX – the import is not possible because the implementation is very different. In TwinCAT 3.1 a corresponding warning is given and a corresponding manual reproduction is necessary.			

Programming languages	ST, IL, FBD	No restrictions
	LD	<p>TwinCAT 3.1 imports function blocks with parallel branches so that the part before the branch is repeated for each branch. This corresponds to the generated code that TwinCAT 2.x generates for parallel branches.</p> 
	SFC	<ul style="list-style-type: none"> • Step variables explicitly declared by the user must be declared locally in the SFC editor. You must not declare them as VAR_INPUT, VAR_OUTPUT or VAR_INOUT, because TwinCAT 3.1 cannot automatically adapt the calls. Explanation: Steps no longer use boolean variables for the management of the internal states in TwinCAT 3.1, but also structures of the type SFCStepType. • Identifier: The following identifiers may not begin with an underscore: <ul style="list-style-type: none"> ◦ Names of IEC actions in the tree ◦ Variables that are called in an IEC association list ◦ Names of programmed-out transitions <p>Explanation: Implicit variables that TwinCAT 3.1 creates for actions are given an underscore as a prefix in TwinCAT 3.1. An invalid identifier with a double underscore would result.</p>
	CFC	<ul style="list-style-type: none"> • Large function blocks: The layout of large function blocks may suffer a loss of quality through being imported; the function block boxes may overlap significantly. • Macros: Macros cannot be imported.

See also:

- TC3 User Interface documentation: [Command Add Existing Item \(Project\) \[▶ 863\]](#)

4.6 Configuring a PLC project

You can configure your TwinCAT PLC project in the following dialogs and views:

- TwinCAT options: General project-independent settings for the behavior of editors
- PLC project properties: Project-related properties, information and settings regarding compiler, etc.
- PLC project settings: General project settings

See also:

- TC3 User Interface documentation: [Command Properties \(PLC project\) \[▶ 906\]](#)
- TC3 User Interface documentation: [Command Options \[▶ 966\]](#)
- TC3 User Interface documentation: [PLC project settings \[▶ 924\]](#)

4.7 Using global data types

If data types are to be used not only within a PLC project but also across PLC projects, these data types can be converted to global data types. Global data types are managed in the type system of the System Manager and automatically made available to the PLC project.

Until build 4026, the global data types are passed to the PLC project using a "super global" flag. In this way, they are available not only in the PLC project itself, but also in the libraries used in it.

With build 4026, virtual data type libraries are automatically created and added to the PLC project and the libraries used in it. On the top level it is the Tc3_GlobalTypes, which brings the actual data type libraries below. All datatypes with namespace are grouped into a corresponding library with this namespace, all others are added under the library Tc3_GlobalTypes_Global.

5 Exporting and transferring a PLC project

Export and import functions are available to you for the exchange of data from TwinCAT projects with other programs.

An exchange of TwinCAT projects between TwinCAT development systems takes place by means of a copy of the project file (*.project) or the project archive (*.projectarchive).

5.1 Exporting and importing a PLC project

TwinCAT offers commands for exporting and importing objects into or out of a file. There are several options here:

- Export to or import from a ZIP file (*.zip)
You can use this format to export and import several files (including non-TwinCAT project files) including the paths.
- Export to a project archive (*.zip)
You can use this format to export an entire project (including the libraries used) to a ZIP archive.
- Export to or import from an XML file in the PLCopen format (*.xml)
You can use this format to exchange information with other programs (for example program editors or documentation tools). PLCopen XML defines a subset of the elements known in TwinCAT. 100% compatibility is therefore not ensured.



The ZIP export or import takes place in the same way as the PLCopenXML export or import described here.

Exporting a project

- ✓ A project is open in TwinCAT.
 - 1. Mark the PLC objects or the PLC project that you wish to export.
 - 2. In the context menu, select the command **Export PLCopenXML...**
 - 3. In the dialog **Export PLCopenXML file**, select the memory location or the file name where you wish to save the exported file.
 - 4. Confirm with **Save**.
- ⇒ The exported file is available to you at the selected memory location.

Importing a project

- ✓ A project is open in TwinCAT.
 - 1. Mark the PLC project or the folder in the PLC project into which the file is to be imported.
 - 2. In the context menu, select the command **Import PLCopenXML...**
 - 3. In the dialog **Import PLCopenXML file**, select the file that you wish to import.
 - ⇒ A dialog opens and shows the objects in a tree structure that can be inserted at this point.
 - 4. Select the objects and click on **Open**.
- ⇒ The objects are inserted in the existing project tree.

See also:

- TC3 User Interface documentation: Command Export to ZIP
- TC3 User Interface documentation: Command Import from ZIP
- TC3 User Interface documentation: Command Export PLCopenXML
- TC3 User Interface documentation: Command Import PLCopenXML
- TC3 User Interface documentation: [Dialog Options - PLCopenXML \[► 976\]](#)
- TC3 User Interface documentation: [Dialog Options - ZIP export/import \[► 995\]](#)

5.2 Transferring a PLC project

If you transfer a project to a different computer and wish to connect to the same PLC from there without an online change or download being necessary, note the following points:

- Make sure that the project only demands fixed versions of libraries (exception: interface libraries), visualization profile and compiler.
- Make sure that the boot project is up to date.

Create a project archive and unpack it on the other computer or check the project into your source code administration system.

Transferring a project to a different system

- ✓ On a computer "PC1", the project is opened that you wish to transfer to a different computer "PC2", from where you want to connect again to the same controller.
 - ✓ Only libraries with fixed versions are integrated in the project (exception: pure interface libraries). To check this, open the library manager. If, instead of a fixed version ID, a "*" stands next to a library entry, the library is not integrated with a fixed version. (See section "[Using libraries \[► 266\]](#)")
 - ✓ The opened PLC project is the same as the one that is currently in use on the PLC. This means that the "boot project" is identical to the project in the programming system. If the diskette symbols of the PLC project and PLC objects in the PLC project tree are red, this means that the project or a PLC object has been changed, but not yet saved. In this case the PLC project and the boot project may not match. Save the changes and generate a new boot project. In order to explicitly generate a boot project, select the command **Activate Boot Project** in the context menu of the PLC node. Log in to the PLC with the command **Login** and start the execution with the command **Start**. The project is now running on the PLC to which you wish to connect again from the same project on PC2.
1. Select the command **Save <TwinCAT project name> as Archive...** in the context menu of the PLC project in order to generate a project archive or check the entire project into your source code administration system.
 2. Transfer the project archive to PC2 and extract it or load the latest version from your source code administration system into PC2.
 3. Now open the TwinCAT project or integrate the PLC in a new TwinCAT 3 project.
 4. Compile the project.
 5. Log in to the PLC again.
- ⇒ TwinCAT does not demand an online change or download. The project runs.

See also:

- TC3 User Interface documentation: [Command Activate Boot Project](#)
- TC3 User Interface documentation: [Command Login \[► 957\]](#)
- TC3 User Interface documentation: [Command Save <TwinCAT project name> as Archive... \[► 1064\]](#)
- TC3 User Interface documentation: [Command Start \[► 958\]](#)

6 Localizing the PLC project

You can display your project in different languages if you create and integrate localization files. The localization files correspond to those of the GNU “gettext” system. The localization template files are *.pot files (Portable Object Template) from which localization files are created in *.po format (Portable Object) after the translation.



Although the project can be presented in different languages, editing is only possible in the original version.

You configure which categories of text information contained in the project you want to localize. Then you export these texts into a translation template. This template is a file of the format pot (for example "project_1.pot"). The template serves as a basis for creating localization files in po format (e.g. "de.po", "en.po", "es.po"), using a suitable external translation tool, or manually using a simple text editor. You can then import the po files back into TwinCAT and use them for the localization. The commands for managing the project localization can be found in the menu **PLC > Project Localization**.

Generating a localization template

✓ A project is open.

1. Select the command **Create Localization Template** in the menu **PLC > Project Localization**.

⇒ The **Generating a localization template...** dialog opens.

2. Select the text information categories you want to include in the localization template.

3. "Position information" can also be included in the template. For each text to be translated you specify its location in the project. Here you can select whether only the first instance of the text that was found, all instances or none should be displayed in the translation template.

4. Click **Create**.

⇒ The dialog for saving a file in pot format opens. Save the localization template. You can then edit the file in a translation tool and create <Language>.po localization files in the desired languages.

Format of the localization template (file *.pot))

The first line shows which text categories were selected for translation when the template was created:

Example:

#: Content:Comments|Identifiers|Names|Strings: All four categories were selected.

For each text to be translated, this is followed by a section in the form shown in the example below:

Example:

```
#: D:\Projects\p1.project\Project_Settings:1
msgid "Project Settings"
msgstr ""
```

Line 1: Position information as source code reference: This is only shown if it was configured when the translation file was generated.

Line 2: Untranslated text as entry msgid. Example: msgid "Project Settings".

Line 3: Placeholder for the translation: msgstr "". The translation in the respective language must then be inserted between the quotation marks in the po file.

Format of the localization file (file *-<Language>.po)

You can create a po file using a translation tool or manually using a neutral text editor based on the pot file. You could rename the pot file to a po file and then edit it according to the standard po format. Be sure to specify the language in the metadata of the file in the form of the usual language tag.

Example: "Language: de" for German.

Enter the translations of the individual texts into the msgstr "" entries between the quotation marks.

Example:

```
"Language: de\n"  
#: Content:Names  
#: D:\projects\pl.project\Project_Settings:1  
msgid "Project Settings"  
msgstr "Projekteinstellungen"
```

Importing the localization files (localizing the project)

- ✓ Localization files `<language>.po` were created for your project, based on the `*.pot` localization template. The project is open.
- 1. Select the command **Manage Localization ...** in the menu **PLC > Project Localization**
- 2. Click **Add**.
 - ⇒ The **Open Localization File** dialog for selecting a po file from the file system appears.
- 3. Select one of the localization files, for example "`<project name>-de.po`".
- 4. The dialog closes, and the affected texts appear in the corresponding language in the project. For example, if you have entered the translation `msgstr "Hauptprogramm"` in the German localization file for the function block name MAIN, the object name **Hauptprogramm** appears in the PLC project tree.
- 5. Import the localization files for the other languages, for which translations exist, in the same way.

Changing the localization (adding and removing localization files)

- ✓ All the required languages are stored in the project after importing the corresponding po files.
- ✓ The project is open.
- 1. Select the command **Manage Localization ...** in the menu **Project > Project Localization**.
 - ⇒ The **Manage Localization** dialog opens. All stored localization files `*-<language>.po` and the entry **<original version>** appear under **Files**.
- 2. Select the required language and click **Switch Localization**.
 - ⇒ The project appears in the selected language. If you select **<original version>** the project appears in the original, non-localized version and can be edited again.

Optional: specifying a standard localization (Toggle Localization)

Select one of the available localization and enable the option **Standard Localization**.

Use the command **Toggle Localization** in the menu **PLC > Project Localization** to switch the localization between the standard localization and the original version. By default, the command is also available via the

 button in the toolbar.

See also:

- TC3 User Interface documentation: [Command Create Localization Template \[► 964\]](#)
- TC3 User Interface documentation: [Command Manage Localization \[► 965\]](#)
- TC3 User Interface documentation: [Command Toggle Localization \[► 965\]](#)

7 Programming a PLC project

To create an application program that can run on the controller, fill POU's with declarations and implementation code (source code), link the I/Os of the controller to program variables, and configure a task assignment. After checking and debugging, the compiler creates the program code that can be loaded onto the controller.

Programming of the project blocks (POUs) is supported by the programming language editors and certain further functionalities such as pragmas and refactoring, and the application of ready-made function blocks from TwinCAT 3 PLC libraries.

Tools exist for syntax checking and code analysis, for creating data persistence and for encrypting the program code that is loaded onto the controller.

See also:

- [Your first TwinCAT 3 PLC project \[▶ 24\]](#)

7.1 Assigning identifiers

Identifiers are the names of variables and programming objects such as programs, function blocks, methods etc. and names of other PLC project objects. There are certain rules that you must follow when assigning identifiers. In addition, there are recommendations that serve to make the identifiers consistent and meaningful.

Variable identifiers are assigned at the variable declaration stage. You can change these identifiers in the declaration part of the programming object. The identifiers for programming and other objects are assigned in the dialog when the respective object is added. You can change the identifier of an existing PLC project object in the Properties window of the object. Identifiers of objects, which can only exist once in each PLC project, cannot be modified. These include the References and RecipeManager identifiers.

See also:

- Reference Programming > [Identifiers \[▶ 844\]](#)
- TwinCAT 3 Programming conventions > Identifier/name

7.2 Declaring variables

Variable Declaration

You can declare variables in the following places:

- Declaration part of a programming object
- GVL editor

The dialog **Auto Declare** supports you with the variable declaration.

Syntax

```
( <pragma> ) *
<scope> ( <type qualifier> ) ?
    <identifier> (AT <address> ) ? : <data type> ( := <initial value> ) ? ;
END_VAR
```

<pragma> (optional)	Pragma (no at all, once or several times) The properties of one or several variables can be influenced by adding a pragma.	See also • Using pragmas [► 137]
<scope>	Scope • VAR • VAR_CONFIG • VAR_EXTERNAL • VAR_GLOBAL • VAR_INPUT • VAR_INST • VAR_IN_OUT • VAR_OUTPUT • VAR_STAT • VAR_TEMP	See also • Variables [► 682]
<type qualifier> (optional)	Type qualifier • CONSTANT • RETAIN • PERSISTENT	See also • Variable types - attribute keywords [► 695]
<identifier>	Identifier, variable name It is important to follow the rules listed in section “Identifiers” when assigning the identifiers. Additional harmonization conventions can be found in section “Identifiers/Names”.	See also • Identifier [► 844] • Identifier/name
AT <address> (optional)	Assignment of an address in the input, output, or flag memory area (I, Q, or M) Example: AT %I*, AT %Q*	See also • AT-Declaration [► 69] • Addresses [► 754]
<data type>	Data type • <elementary data type> • <user defined data type> • <function block >	See also • Data types [► 756]
<initial value> (optional)	Initial value • <literal value> • <identifier> • <expression>	See also • Operands [► 744] • ST Expressions [► 625]
(...)?	Optional	
(...)*	Optional repetition	

Variable initialization

The standard initialization value for all declarations is 0. In the declaration part you can specify user-defined initialization values for each variable and each data type.

The user-defined initialization starts with the allocation operator := and consists of a valid expression in the programming language ST (Structured Text). The initialization value is thus defined with the aid of constants, other variables or functions. If you use a variable, this also has to be initialized.

Example 1:

```

VAR
  nVar1   : INT := 12;           // initialization value 12
  nVar2   : INT := 13 + 8;      //
initialization value defined by an expression of constants
  nVar3   : INT := nVar2 + F_Fun(4); //
initialization value defined by an expression that contains a function call; notice the order!
  pSample : POINTER TO INT := ADR(nVar1); //
not described in the standard IEC61131-3: initialization value defined by an adress function; Notice
: the pointer will not be initialized during an Online Change
END_VAR

```

Example 2:

In the following sample, an input variable and a property are initialized by a function block that has an **FB_init** method [[► 850](#)] with an additional parameter.

Function block FB_Sample:

```

FUNCTION_BLOCK FB_Sample
VAR_INPUT
  nInput      : INT;
END_VAR
VAR
  nLocalInitParam : INT;
  nLocalProp      : INT;
END_VAR

```

Method FB_Sample.FB_init:

```

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)
  nInitParam   : INT;
END_VAR
nLocalInitParam := nInitParam;

```

Property FB_Sample.nMyProperty and the associated Set function:

```

PROPERTY nMyProperty : INT
nLocalProp := nMyProperty;

```

Program MAIN:

```

PROGRAM MAIN
VAR
  fbSample : FB_Sample(nInitParam := 1) := (nInput := 2, nMyProperty := 3);
  aSample  : ARRAY[1..2] OF FB_Sample[(nInitParam := 4), (nInitParam := 7)]
           := [(nInput := 5, nMyProperty := 6), (nInput := 8, nMyProperty := 9)];
END_VAR

```

Initialization result:

- fbSample
 - nInput = 2
 - nLocalInitParam = 1
 - nLocalProp = 3
- aSample[1]
 - nInput = 5
 - nLocalInitParam = 4
 - nLocalProp = 6
- aSample[2]
 - nInput = 8
 - nLocalInitParam = 7
 - nLocalProp = 9

See also:

- Reference Programming > [Data types](#) [[► 756](#)]

- Reference Programming > [Variable types and special variables \[► 682\]](#)
- Reference Programming > Operands > [Addresses \[► 754\]](#)
- Reference Programming > [Identifier \[► 844\]](#)
- TwinCAT 3 programming conventions > Identifier/name

7.2.1 AT-Declaration

To bind a project variable to a flexible (*) or direct address, you can specify the address when declaring the variables. By using an appropriate variable name, you can give the address a meaningful name.



Automatic addressing

It is recommended not to use direct addressing for allocated variables, but to use the * placeholder instead.

If the placeholders * (%I*, %Q* or %M*) are used, TwinCAT automatically performs flexible and optimized addressing.

Syntax:

```
<identifier> AT <address> : <data type>;
```

If an address is specified, the position in the memory and the size are expressed via special strings. An address is marked with the percent sign %, then follows the memory area prefix, the optional size prefix and the memory position.

```
%<memory area prefix> ( <size prefix> )? <memory position>
```

```
<memory area prefix> : I | Q | M
<size prefix>       : X | B | W | D
<memory position>  : * | <number> ( .<number> ) *
```

Memory area prefix

I	Input memory area for inputs For physical inputs via input drivers ("sensors")
Q	Output memory area for outputs Physical outputs via output drivers ("actuators")
M	Flag memory area

Size prefix

X	Single bit
B	Byte (8 bits)
W	Word (16 bits)
D	Double word (32 bits)

Examples:

```
IbSensor1 AT%I* : BOOL;
IbSensor2 AT%IX7.5 : BOOL;
```



If you do not explicitly specify a single bit address, Boolean variables are allocated byte by byte. Example: a value change of bVar AT %QB0 concerns the area from QX0.0 bis QX0.7.

If you assign a variable to an address, you must observe the following:

- In the implementation editor, you cannot write-access variables that are assigned to an input with direct addressing. This leads to a compiler error.
- If you use AT declarations with direct addressing to structure or function block components, all instances use the same memory. This corresponds to the use of [static variables \[► 687\]](#) in classic programming languages such as "C".
- The memory layout of structures is likewise dependent on the target system.



Valid addresses

The keyword AT must follow a valid address. Further information on this can be found in the section Reference Programming > Operands > [Addresses \[► 754\]](#). Pay attention to possible overlaps in the case of byte addressing mode.

Examples

Variable declarations:

lbSensor AT%I* : BOOL;	In the address specification, the placeholder * is specified instead of the memory position. This enables TwinCAT to perform flexible and optimized addressing automatically.
InInput AT%IW0 : WORD;	Variable declaration with address specification of an input word
ObActuator AT%QB0 : BOOL;	Boolean variable declaration Note: for Boolean variable one byte is allocated internally if no single bit address is specified. A value change of ObActuator consequently affects the range from QX0.0 to QX0.7.
lbSensor AT%IX7.5 : BOOL;	Boolean variable declaration with explicit specification of a single bit address. Only input bit 7.5 is read during access.

Other addresses:

%QX7.5	Single bit address of the output bit 7.5
%Q7.5	
%IW215	Word address of the input word 215
%QB7	Byte address of the output byte 7
%MD48	Address of a double word at memory location 48 in the flag area
%IW2.5.7.1	The interpretation depends on the current controller configuration (see below)



See also:

- Reference Programming > Operands > [Addresses \[► 754\]](#)

7.2.2 Using the Declaration Editor

The declaration editor is used for declaring variables in variable lists and POU's.

If the declaration editor is used in conjunction with a programming language editor, it appears as declaration part in the upper part of a POU window.

The declaration editor offers two possible views: textual () or tabular (). In the dialog **Tools > Options > TwinCAT > PLC Environment > Declaration editor** you can define whether either only the text-based view or only the tabular view is available, or whether the user can choose between the two views via the buttons on the right of the editor window.

See also:

- Reference Programming: [Declaration Editor \[► 622\]](#)

Declarations via the textual declaration editor


- ✓ A programming object (POU or GVL) of a project is open. The focus is on the textual declaration editor.

1. Enter the variable declarations in the correct syntax. Use **[F2]** or the **Input Assistant** option in the context menu to open the **Input Assistant** dialog for selecting the data type or a keyword. Use the **Auto Declare** command in the context menu to open the **Auto Declare** dialog.
 - ⇒ When variables are declared, keywords are automatically corrected and highlighted.

See also:

- [Using the Auto Declare dialog \[▶ 71\]](#)
- TC3 User Interface documentation: [Command Input Assistant \[▶ 873\]](#)
- TC3 User Interface documentation: [Command Auto Declare \[▶ 875\]](#)

Declarations via the tabular declaration editor

- ✓ A programming object (POU or GVL) of a project is open. The focus is on the tabular declaration editor.
1. Click the button  in the declaration header or select the **Auto Declare** command in the context menu to open the **Auto Declare** dialog.
 - ⇒ TwinCAT inserts a new row for a variable declaration, and the input field for the variable name opens.
 2. Enter a valid variable identifier.
 3. Open the other fields of the declaration line by double-clicking as required, and select the required information from the selection lists or via the displayed dialogs.
 - ⇒ The correct syntax is used automatically when variables are declared.

See also:

- TC3 User Interface documentation: [Command Auto Declare \[▶ 875\]](#)

7.2.3 Using the Auto Declare dialog

- ✓ A programming object (POU or GVL) of a project is open.
1. Select the **Auto Declare** command in the **Edit** menu or in the context menu of the editor.
 - ⇒ The dialog **Auto Declare** opens.
 2. Select the required validity range for the variable from the **Scope** selection list.
 3. Enter a variable name in the **Name** field.
 4. Select the required data type from the **Type** selection list.
 5. To use an initialization value that differs from the standard initialization value, enter an initialization value for the variable.
 6. Finish your entries by clicking **OK**.
 - ⇒ TwinCAT lists the newly declared variable in the declaration part of your programming object.





You can use pragmas in the declaration part to influence the processing of the declaration by the compiler. (See section "[Using pragmas \[▶ 137\]](#)")

See also:

- TC3 User Interface documentation: [Command Auto Declare \[▶ 875\]](#)

7.2.4 Declaring an array


- ✓ A programming object (POU or GVL) of a project is open.
1. Select the **Auto Declare** command in the **Edit** menu or in the context menu of the editor.
 - ⇒ The **Auto Declare** dialog opens.
 2. Select the required scope for the array from the **Scope** selection list.
 3. Enter an identifier for the array in the **Name** input field.

4. Click on  next to the **Type** input field and select the entry **Array Wizard** from the selection menu.
5. Enter the lower and the upper index bounds for the first dimension of the array in the **Dimension 1** input fields, for example: 1 and 3.
 - ⇒ The **Result** field shows the 1st dimension of the array, for example: ARRAY [1..3] OF ?.
6. In the input field **Base Type** enter the data type of the array, either directly or via the **Input Assistant** or the **Array Wizard** (button ) , for example: DINT.
 - ⇒ The **Result** field shows the data type of the array, for example: ARRAY [1..3] OF DINT.
7. Define dimensions 2 and 3 of the array according to steps 5 and 6, for example: dimension 2: 1 and 4, dimension 3: 1 and 2.
 - ⇒ The **Result** field shows the array with the defined dimensions: Array [1..3, 1..4, 1..2] OF DINT. The array consists of $3 * 4 * 2 = 24$ elements.





For a variable length array, declare the dimension bounds with the asterisk placeholder *. Arrays of variable length are only allowed in VAR_IN_OUT declarations of function blocks, methods or functions.

Example for a two-dimensional array of variable length: aVariableLength : ARRAY [* , *] OF INT;

8. Click **OK**.
 - ⇒ In the **Auto Declare** dialog, the **Type** field shows the array.
9. To change the initialization values of the array, click  next to the **Initialization Value** input field.
 - ⇒ The **Initialization Value** dialog opens.
10. Select the row of the array element, whose initialization value you want to change. Example: select array element [1, 1, 1].
11. Enter the required initialization value in the input field below the list and click on **Apply value to selected lines**, for example: "Value 4".
 - ⇒ TwinCAT shows the modified initialization value for the selected row.
12. Click **OK**.
 - ⇒ TwinCAT shows the initialization values of the array in the **Initialization** field of the **Auto Declare** dialog, for example: [4, 23(0)].
13. If required, you can enter a comment in the input field (optional).
14. Click **OK** to complete the array declaration.
 - ⇒ TwinCAT adds the array declaration to the declaration part of the programming object.

See also:

- TC3 User Interface documentation: [Command Auto Declare](#)  [875](#)
- Reference Programming > [ARRAY](#)  [773](#)

7.2.5 Declaring global variables

Global variable lists are used for declaring and editing global variables. Parameter lists are used for declaring global constants in libraries.

See also:

- Reference Programming > [Remanent variables - RETAIN, PERSISTENT](#)  [691](#)

7.2.5.1 Object Global Variable List


Symbol: 

A global variable list is used for declaring, editing and displaying global variables. If you add a GVL to the project, the variables apply to the whole project.

Object Creating a global variable list

1. Select the folder **GVLs** in the **Solution Explorer** in the PLC project tree.
2. In the context menu select the command **Add > Global Variable List**.
3. Enter a name and click **Open**.
 - ⇒ TwinCAT adds the GVL to the PLC project tree and opens it in the editor. You can define the global variables between the keywords `VAR_GLOBAL` and `END_VAR`.


Defining global variables

- ✓ A GVL is open in the editor.
1. Enter the variable declarations in the correct syntax, or select the **Auto Declare** command in the **Edit** menu or in the context menu of the editor.
 - ⇒ The **Auto Declare** dialog opens. The entry `VAR_GLOBAL` is selected in the **Scope** selection list.
 2. In the **Name** field enter a name for the global variable.
 3. Select a data type in the **Type** selection list.
 4. If a variable is to have an initialization value that differs from the standard initialization value, click on  next to the initialization value field.
 - ⇒ The **Initialization Value** dialog opens.
 5. Double-click on the **Initialization** cell of your variable and enter the required valid value.
 6. Click on **OK**.
 - ⇒ The initialization value is shown in the **Auto Declare** dialog.
 7. Activate one of the flags, if required.
 8. Confirm your inputs by clicking **OK**.
 - ⇒ TwinCAT adds the declared variable in the GVL. The global variable is available in the whole PLC project.

See also:

- TC3 User Interface documentation: [Command Auto Declare](#) [▶ 875]

7.2.5.2 Object Parameter List

Symbol: 

If you want to configure global constants, which are provided by the library, at a later stage in a PLC project, you can define them in a parameter list. A parameter list is a special type of a global variable list.

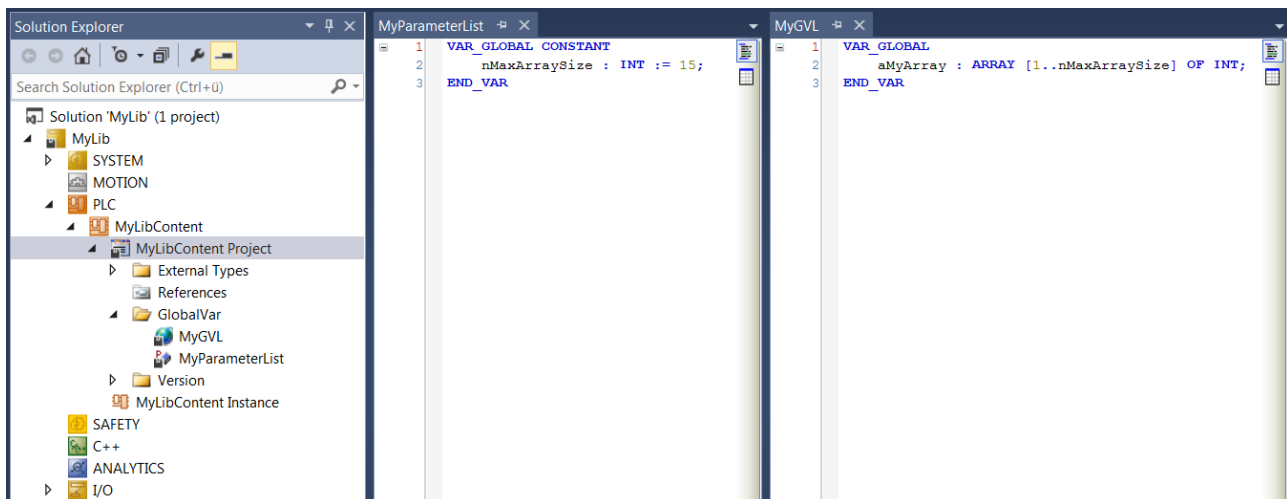
Creating an object Parameter List

- ✓ A library project is open.
1. In the **Solution Explorer**, select the PLC project <Projectname.project> in the PLC project tree.
 2. In the context menu select the command **Add > Parameter List...**
 3. Enter a name and click **Open**.
 - ⇒ TwinCAT adds the parameter list to the PLC project tree and opens it in the editor. You can define the global constants between the keywords `VAR_GLOBAL CONSTANT` and `END_VAR`.

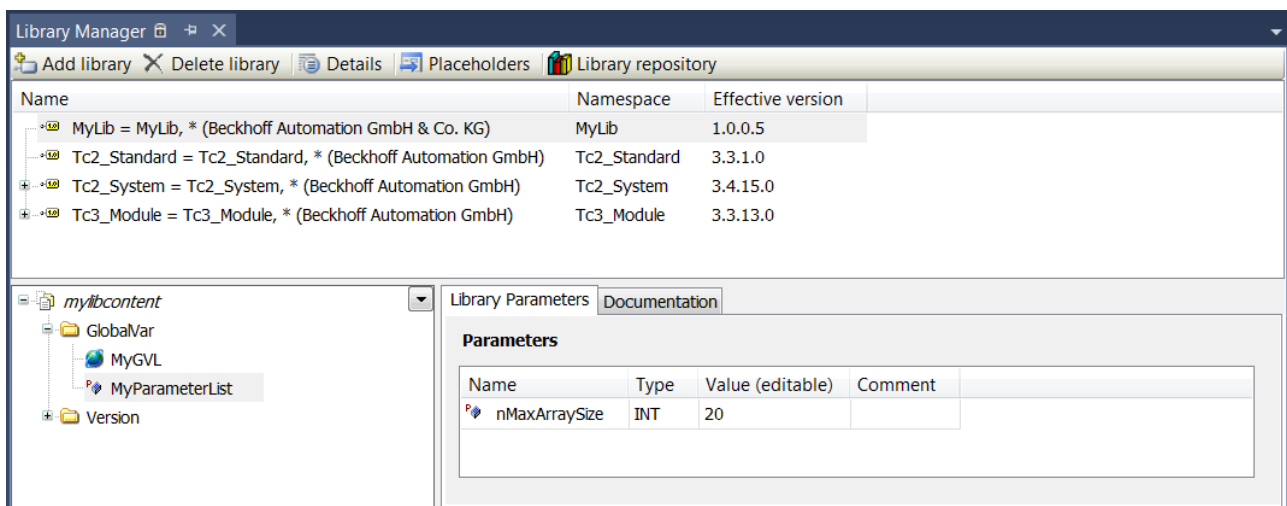
Example:

A MyLib library provides an `ARRAY` variable `aMyArray`, whose size is defined by the global constant `nMaxArraySize`. The library is integrated in various PLC projects. The PLC projects use different array sizes, and the global constant from the library should be overwritten by a project-specific value.

The global constant `nMaxArraySize` is defined within a parameter list when the MyLib library is created. First, a **Parameter List** object with the name `MyParameterList` is added to the PLC project with the command **Add** in the context menu. The variable `nMaxArraySize` is declared in the editor for the object.



The library MyLib is integrated in a PLC project. The library manager is opened to replace the value of the global constant with a project-specific value. The library is selected in the upper section. The function block tree with the parameter list MyParameterList is displayed in the lower section. The parameter list is selected. The **Library Parameters** tab with the declarations contained in the parameter list opens at the bottom right. The value of the global constant nMaxArraySize to be edited is selected in the column **Value (editable)**. Pressing the space bar opens an input field, where the required new value for nArraySize can be entered. When the input field is closed, the value is applied to the local scope of the library.



Exporting and importing parameters



Available from TwinCAT 3.1 Build 4026

Exporting parameters:

- ✓ A library with parameter list is selected in the library management.
 - 1. In the library management, select the parameter list.
 - 2. Select the command **Export Library Parameters...** from the context menu
 - 3. Select a folder, enter a file name and click **Save**.
- ⇒ The edited values from the parameter list are saved in a csv file.

Importing parameters:

- ✓ A library with parameter list is selected in the library management.
- 1. In the library management, select the parameter list.
- 2. Select the command **Import Library Parameters...** from the context menu



3. Select a csv file with saved parameters and click **Open**.
⇒ The values stored in the csv file are inserted into the parameter list.

7.3 Creating user-specific data types

In addition to the standard data types, you can define your own data types such as structures, enumerations, references and unions. These are created as data type objects (DUT = data unit types).

7.3.1 Object DUT

Symbol:

-  for a DUT without text list support
-  for a DUT of type enumeration with text list support

A DUT (data unit type) describes a user-specific data type.





Creating an object DUT

1. Select a folder in the **Solution Explorer** in the PLC project tree.
2. In the context menu select the command **Add > DUT...**
⇒ The dialog **Add DUT** opens.
3. Enter a name and select a data type.
4. Click on **Open**.
⇒ The DUT is added to the PLC project tree and opens in the editor.

Dialog Add DUT

Name	Name of the new DUT
------	---------------------

Data type

Structure	<p>Creates an object that declares a structure that combines several variables with different data types into one logical unit.</p> <p>The variables declared within the structure are referred to as components.</p> <p><input checked="" type="checkbox"/> Advanced: The structure extends an existing structure with additional components. Specify an existing structure in the input field next to it. The components of the existing structure are automatically available in the new one.</p> <p>(See also: Structure [▶ 778])</p>
Enumeration	<p>Creates an object that declares an enumeration that combines multiple integer constants into a logical unit</p> <p>The constants declared within an enumeration are also called enumeration values.</p> <p><input type="checkbox"/> Text list support: An enumeration without text list support is created. The DUT object appears in the PLC project tree in the Solution Explorer with the following symbol:</p> <p></p> <p><input checked="" type="checkbox"/> Text list support: The text list allows you to locate the names of the enumeration values. The DUT object appears in the PLC project tree in the Solution Explorer with the following symbol:</p> <p></p> <p>You can output the localized texts in a visualization, for example. Then, in the text output of a visualization element, the symbolic enumeration values appear in the current language instead of the numeric enumeration values. When a text list-supported enumeration variable is entered in the Text variable property of a visualization element, the supplement <enumeration name> is added.</p> <p>The buttons at the right edge of the editor can be used to switch between Textual view () and Localization view (text list) ().</p> <p>Sample: The variable MAIN.eVar of type E_myEnum is used. E_myEnum is a text list-supported DUT. The entry in the properties editor then looks as follows: MAIN.eVar <E_myEnum>. If the enumeration name is changed in the PLC project, a prompt appears with the question whether TwinCAT should update the affected visualizations accordingly.</p> <p>On an existing enumeration object, the text list support can be added or removed at any time afterwards: The commands Add text list support or Remove text list support in the context menu of the object can be used for this purpose.</p> <p>(See also: Enumerations [▶ 781])</p>
Alias	<p>Creates an object that declares an alias that is used to declare an alternate name for a base type, data type, or function block.</p> <p>You can enter the Base Type directly or select it via the input assistant or the array wizard.</p> <p>(See also: Alias [▶ 784])</p>
Union	<p>Creates an object that declares a union that combines multiple components, usually of different data types, into a single logical unit.</p> <p>All components have the same offset, so they occupy the same storage space. The memory footprint of a union is determined by the memory footprint of its “largest” component.</p> <p>(See also: UNION [▶ 784])</p>

Declaring a DUT

Syntax:

```
TYPE <identifier> : <data type declaration with optional initialization>
END_TYPE
```

The syntax of the data type declaration depends in detail on the selected data type (e.g. structure or enumeration).

Samples:

Declaration of a structure

The following section shows two DUTs, which define the structures ST_Struct1 and ST_Struct2. The structure ST_Struct2 extends the structure ST_Struct1, which means that ST_Struct2.nVar1 can be used to access the variable nVar1.

```
TYPE ST_Struct1 :
STRUCT
    nVar1 : INT;
    bVar2 : BOOL;
END_STRUCT
END_TYPE

TYPE ST_Struct2 EXTENDS ST_Struct1 :
STRUCT
    nVar3 : DWORD;
    sVar4 : STRING;
END_STRUCT
END_TYPE
```

Declaration of an enumeration

```
TYPE E_TrafficSignal :
(
    eRed,
    eYellow,
    eGreen := 10
);
END_TYPE
```

Declaration of an alias

```
TYPE T_Message : STRING[50];
END_TYPE
```


Declaration of a union of components with different data types

```
TYPE U_Name :
UNION
    fA : LREAL;
    nB : LINT;
    nC : WORD;
END_UNION
END_TYPE
```

7.4 Creating programming objects

You can add various programming objects to your PLC project, in which you can write and structure the source code for your control program.

7.4.1 Object POU

Symbol: 

An object of type **POU** is a program organization unit in a TwinCAT PLC project, as defined in the IEC 61131-3 standard.

The following POU types can be added to the PLC project:

- [Function \[► 81\]](#)
- [Function block \[► 84\]](#)
- [Program \[► 86\]](#)

In addition, you can add the following programming objects to these objects:

- [Action \[► 87\]](#)
- [Transition \[► 88\]](#)
- [Method \[► 90\]](#)
- [Property \[► 96\]](#)

Certain POUs can call other POUs. Recursions are not allowed.

When POUs are called via the namespace, TwinCAT searches the project for the calling POU according to the following order:

1. Current application
2. Library manager for the current application

Creating an object POU

1. Select a folder in the **Solution Explorer** in the PLC project tree.
2. In the context menu select the command **Add > POU...**
 - ⇒ The **Add POU** dialog opens.
3. Enter a name and select a type and the implementation language.
4. Click on **Open**.
 - ⇒ The POU is added to the PLC project tree and opened in the editor. The editor consists of the declaration editor at the top and the implementation part at the bottom. Depending on the implementation language, the **Toolbox** view may automatically become active, in which suitable elements, operators and function blocks are provided.

Dialog Add POU

Name	Name of the POU
Implementation language	Selection list for the implementation language of the POU

Type

Program	
Function block	<ul style="list-style-type: none"> • <input checked="" type="checkbox"/> Advanced: Enter or select a predefined function block for object-oriented programming. This is specified with keyword EXTENDS in the function block declaration. • <input checked="" type="checkbox"/> Implemented: Enter or select an interface for object-oriented programming. This is specified with keyword IMPLEMENTS in the function block declaration. When the POU is created, all methods defined via the interface are created. See also Automatic creation of interface elements in a function block [► 84] • <input checked="" type="checkbox"/> Final: Derived access is not allowed. It means that you cannot extend the function block with another function block. This enables you to check whether or not further derivations should be allowed. This enables optimized code generation. • <input checked="" type="checkbox"/> Abstract: Indicates that the function block has a missing or incomplete implementation and cannot be instanced. Abstract function blocks serve exclusively as basic function blocks and the implementation typically takes place in a derived function block. If a non-abstract function block is created, which in turn extends an abstract function block, all methods of the abstract basic function block are added as (non-abstract) methods to the new function block. See also ABSTRACT concept [► 201] • Access modifier <ul style="list-style-type: none"> ◦ PUBLIC: Corresponds to the specification of no access modifier ◦ INTERNAL: Access to the function block is limited to the namespace (the library). • Method implementation language: If you have selected the option Implements, you can use this option to select an implementation language for all method objects, which TwinCAT generates via the implementation of the interface. The method implementation language is independent of the implementation language for the function block.
Function	<p>Not available, if the Sequential Function Chart language (SFC) is selected in the selection list for the implementation language.</p> <p>Return type: Selection list for the data type of the return value</p>

7.4.1.1 Object Function

A function is a POU, which, when executed, returns precisely one data element, and whose call may occur in text-based languages as operator in expressions. The data element may also be an array or a structure.

In the PLC project tree, function POUs have the suffix (FUN). The editor of a function consists of the declaration part and the implementation part.



All data of a function are temporary and are only valid while the function is executed (stack variables). This means that TwinCAT re-initializes all variables that you have declared in a function each time the function is called.



Calls of a function with the same input variable values always return the same output value. Therefore, functions may not use global variables and addresses!

The top line of the declaration part contains the following declaration:

```
FUNCTION <function> : <data type>
```

The input and function variables are declared below.

The output variable of a function is the function name.

i If you declare a local variable in a function as RETAIN, this has no effect. In this case TwinCAT issues a compiler error.

i In TwinCAT 3, you cannot mix explicit and implicit parameter assignments in function calls. This means that you should either only use explicit or only implicit parameter assignments in function calls. The order of the parameter assignments in function calls is irrelevant.

Calling a function

In ST you can use function calls as operands in expressions.

In SFC you can use function calls only within step actions or transitions.

Examples:

Function with a declaration part and one line of implementation code:

```

F_Sample  # X
1  FUNCTION F_Sample : INT
2  VAR_INPUT
3      nVar1 : INT;
4      nVar2 : INT;
5      nVar3 : INT;
6  END_VAR
7  VAR
8  END_VAR
1  F_Sample := nVar1 + nVar2 * nVar3;

```

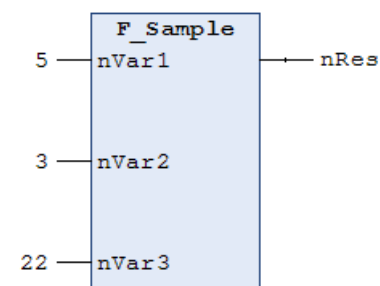
ST:

```
nRes := F_Sample(5,3,22)
```

IL:

1	LD	5
	F_Sample	3
		22
	ST	nRes

FBD:



Functions with additional outputs

According to the IEC 61131-3 standard, functions can have additional outputs. You can declare the additional outputs in the function between the keywords VAR_OUTPUT and END_VAR. The function is called based on the following syntax:

<function> (<function output variable1> => <output variable 1>, <function output variable n> => <output variable n>)

Example:

The function F_Fun is defined with two input variables nIn1 and nIn2. The output variable of the function F_Fun is written to the locally declared output variables nLoc1 and nLoc2.

```
F_Fun(nIn1 := 1, nIn2 := 2, nOut1 => nLoc1, nOut2 => nLoc2);
```

Access to a single element of a structured return type during method/function/property call

The following implementation can be used to directly access an individual element of the structured data type that is returned by the method/function/property when a method, function or property is called. A structured data type is, for example, a structure or a function block.

1. The return type of the method/function/property is defined as "REFERENCE TO <structured type>" (instead of just "<structured type>").
2. Note that with such a return type – if, for example, an FB-local instance of the structured data type is to be returned – the reference operator REF= must be used instead of the "normal" assignment operator :=.

The declarations and the sample in this section refer to the call of a property. However, they are equally transferrable to other calls that deliver return values (e.g. methods or functions).

Sample

Declaration of the structure ST_Sample (structured data type):

```
TYPE ST_Sample :
STRUCT
  bVar : BOOL;
  nVar : INT;
END_STRUCT
END_TYPE
```

Declaration of the function block FB_Sample:

```
FUNCTION_BLOCK FB_Sample
VAR
  stLocal : ST_Sample;
END_VAR
```

Declaration of the property FB_Sample.MyProp with the return type "REFERENCE TO ST_Sample":

```
PROPERTY MyProp : REFERENCE TO ST_Sample
```

Implementation of the Get method of the property FB_Sample.MyProp:

```
MyProp REF= stLocal;
```

Implementation of the Set method of the property FB_Sample.MyProp:

```
stLocal := MyProp;
```

Calling the Get and Set methods in the main program MAIN:

```
PROGRAM MAIN
VAR
  fbSample : FB_Sample;
  nSingleGet : INT;
  stGet : ST_Sample;
  bSet : BOOL;
  stSet : ST_Sample;
END_VAR

// Get - single member and complete structure possible
nSingleGet := fbSample.MyProp.nVar;
stGet := fbSample.MyProp;

// Set - only complete structure possible
IF bSet THEN
  fbSample.MyProp REF= stSet;
  bSet := FALSE;
END_IF
```

Through the declaration of the return type of the property MyProp as "REFERENCE TO ST_Sample" and through the use of the reference operator REF= in the Get method of this property, a single element of the returned structured data type can be accessed directly on calling the property.

```
VAR
    fbSample      : FB_Sample;
    nSingleGet    : INT;
END_VAR

nSingleGet := fbSample.MyProp.nVar;
```

If the return type were only to be declared as "ST_Sample", the structure returned by the property would first have to be assigned to a local structure instance. The individual structure elements could then be queried on the basis of the local structure instance.

```
VAR
    fbSample      : FB_Sample;
    stGet         : ST_Sample;
    nSingleGet    : INT;
END_VAR

stGet         := fbSample.MyProp;
nSingleGet := stGet.nVar;
```

7.4.1.2 Object Function block

A function block is a [POU \[► 78\]](#), which returns one or several values when executed. The values of the output variables and the internal variables are retained until the next execution. This means that the function block may not return the same output values, if it is called repeatedly with the same input variables.

In the PLC project tree, function block POU's have the suffix (FB). The editor of a function block consists of the declaration part and the implementation part.

A function block is always called via an instance, which is a copy of the function block.

In addition to the functionality described in IEC 61131-3, in TwinCAT function blocks can also be used for the following object-oriented programming functionalities:

- Extending a function block ([Extending a function block \[► 191\]](#))
- Implementing interfaces ([Implementing an interface \[► 198\]](#))
- Methods ([Object Method \[► 90\]](#))
- Properties ([Object Property \[► 96\]](#))

The top line of the declaration part contains the following declaration:

```
FUNCTION_BLOCK <access specifier> <function block> | EXTENDS <function block> |
IMPLEMENTS <comma-separated list of interfaces>
```

● 8-byte alignment

i An 8-byte alignment was introduced with TwinCAT 3. Make sure that the alignment is appropriate if data are exchanged as an entire memory block with other controllers or software components (see [Alignment \[► 791\]](#)).

Automatic creation of interface elements in a function block

There are two ways of automatically generating elements of an interface that implements a function block in this function block.

1. If you specify an interface in the field **Implements** in the dialog **Add** when creating a new function block, TwinCAT automatically also adds the methods and properties of the interface to this function block.
2. If an existing function block implements an interface, you can use the command **Implement Interface** in order to generate the interface elements in the function block. The command **Implement Interface** can be found in the context menu of a function block in the project tree.

See also:

- TC3 User Interface documentation: [Command Implement interfaces \[► 1063\]](#)
- [Implementation of an interface \[► 198\]](#)

Calling a function block

The call always takes place via an instance of the function block. If a function block is called, only the values of the respective instance change.

Declaration of the instance:

```
<instance> : <function block>;
```

A variable of the function block is accessed as follows in the implementation part:

```
<instance>.<variable>
```

- From outside the function block instance, you can only access input and output variables of a function block, not the internal variables.
- Access to a function block instance is limited to the POU in which the instance is declared, unless you have declared the instance globally.
- You can assign the desired values to the function block variables when you call the instance.

Sample:

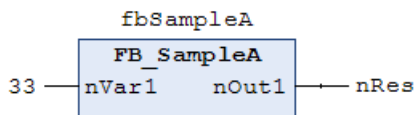
The function block FB_SampleA has the input variable nVar1 of type INT and the output variable nOut1. In the sample below, the variable nVar1 is called from the MAIN program.

ST:

```
PROGRAM MAIN
VAR
    fbSampleA : FB_SampleA;
END_VAR

fbSampleA.nVar1 := 33; (* FB_SampleA is called and the value 33 is assigned to the variable nVar1 *)
fbSampleA(); (* FB_SampleA is called, that's necessary for the following access to the output variable *)
nRes := fbSampleA.nOut1 (* the output variable nOut1 of the FB1 is read *)
```

FBD:



Assigning variable values during a call:

In the text-based languages IL and ST, you can assign values directly to input and/or output variables when the function block is called.

A value is assigned to an input variable with :=

A value is assigned to an output variable with =>

Sample:

The instance fbTimer of the timer function block is called with assignments for the input variables IN and PT. The output variable Q of the timer is then assigned to the variable bVarA

```
PROGRAM MAIN
VAR
    fbTimer : TOF;
    bIn      : BOOL;
    bVarA    : BOOL;
END_VAR

fbTimer(IN := bIn, PT := t#300ms);
bVarA := fbTimer.Q;
```



If you add a function block instance via the input assistant and the option **Insert with arguments** is enabled in the **Input Assistant** dialog, TwinCAT adds the call with all input and output variables. All you have to add is the desired value assignment. In the above example, TwinCAT adds the call as follows: `CMD_TMR (IN:= , PT:= , Q=>)`.



Using the attribute **'is_connected'** on a local variable, you can determine at the time of the call in the function block instance whether a particular input receives an assignment from outside.

7.4.1.3 Object Program

A program is a POU, which returns one or several values when executed. After a program execution, all values are retained until the program is executed again. The call sequence of the programs within a PLC project is defined in task objects.

In the PLC project tree, the program POUs have the suffix (PRG). The editor of a program consists of the declaration part and the implementation part.

The top line of the declaration part contains the following declaration:

```
PROGRAM <program>
```

Calling a program

Programs and function blocks can call a program. Program calls are not allowed in a function. Programs do not have instances.

If a POU calls a program and the call results in changes to the values of the program, the changes are retained until the next program call. The values of the program are also retained, if the next call occurs from another POU. This differs from a function block call. When a function block is called, only the values of the respective function block instance change. The changes are only relevant, if a POU calls the same instance again.

Alternatively, the input and/or output parameters for a program can also be set directly with the call.

Syntax:

```
<program>(<input variable> := <value>, <output value> => <value>):
```

If you add a program call via the input assistant and the option **Insert with arguments** is enabled in the **Input assistant**, TwinCAT adds input and/or output parameters to the program call, according to the corresponding syntax.

Examples:

IL:

1	CAL	SampleProg (
		nIn1:= 2)
	LD	SampleProg.nOut2
	ST	nRes

With parameter assignment:

1	CAL	SampleProg (
		nIn1:= 2
		nOut2=> nRes)

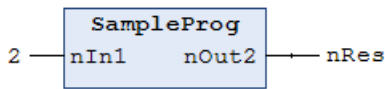
ST:

```
SampleProg();
nRes := SampleProg.nOut2;
```

With parameter assignment:

```
SampleProg(nIn1 := 2, nOut2 => nRes);
```

FBD:



7.4.2 Object Action

Symbol:

An action can be used to implement further program code. You can implement this program code in a language that is different from the basic implementation. The basic implementation is the function block or the program under which you have inserted the action.

An action does not have its own declarations and uses the data of the basic implementation. This means that the action uses the input/output and local variables of the basic implementation.

Creating an object Action

1. Select a function block or a program in the **Solution Explorer** in the PLC project tree.
2. In the context menu select the command **Add > Action...**
 - ⇒ The dialog **Add Action** opens.
3. Enter a name and select an implementation language.
4. Click on **Open**.
 - ⇒ The object is added to the PLC project tree and opens in the editor.

Dialog Add Action

Name	Name of the action
Implementation language	Check box for the implementation language

Calling an action

Syntax:

<program>.<action> or <FB-instance>.<action>

To call an action only within the basic implementation, it is sufficient to specify the action name.

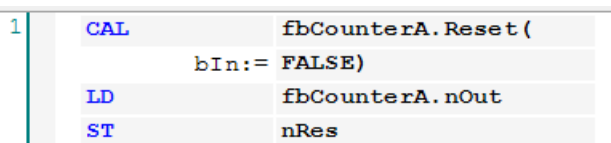
Examples:

Calls a Reset action from another POU. In other words, the call is not made in the basic implementation.

Declaration:

```
PROGRAM MAIN
VAR
    fbCounterA : FB_Counter;
END_VAR
```

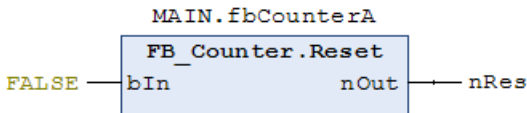
Calls the Reset action from an IL POU:



Calls the Reset action from an ST POU:

```
fbCounterA.Reset(In := FALSE);
nRes := fbCounterA.nOut;
```

Calls the Reset action from an FBD POU:



Actions are common in the implementation language SFC. ([SFC element Action \[▶ 645\]](#))

See also:

- [Object-oriented programming \[▶ 169\]](#)
- [Extending a function block \[▶ 191\]](#)

7.4.3 Object Transition

Symbol:

A transition is used to specify a condition, under which a subsequent step is to become active. A transition condition can have the value TRUE or FALSE. If it is TRUE, the next step is executed. A transition condition can be specified in the following two ways:

- (1) Direct („inline condition“): The standard transition name is replaced with the name of a Boolean variables, a Boolean address, a Boolean constant or a statement with Boolean result (e.g. (i<100) AND b). No programs, function blocks or assignments may be specified.
- (2) A separate transition or property object is used (“multi-use condition“): The standard transition name is replaced with the name of a transition or property object. These objects can be created via the command **Add > Transition...**, which can be found in the context menu (see section “[Object Creating a transition \[▶ 88\]](#)”). This enables multiple use of transitions. Like an “inline condition”, the object may contain a Boolean variable, an address, a constant or a statement, and in addition it may contain multiple statements with any code.

Follow the instructions in section “[Access to VAR IN_OUT variables of the function block \[▶ 89\]](#)”.

Creating an object Transition

1. Select a function block or a program in the **Solution Explorer** in the PLC project tree.
2. In the context menu select the command **Add > Transition...**
 - ⇒ The **Add Transition** dialog opens.
3. Enter a name and select an implementation language.
4. Click on **Open**.
 - ⇒ The object is added to the PLC project tree and opens in the editor.

Dialog Add Transition

Name	Name of the transition
Implementation language	Check box for the implementation language

Calling a transition

Syntax:

In contrast to TwinCAT 2 PLC Control, transition condition is treated like a method call. The input is based on the following syntax:

<transition name> := <transition condition>

or

<transition condition>

If a transition contains multiple statements, you must assign the desired expression to the transition variable (first variant of the syntax).

Samples:

Call of transition Trans1 in an ST POU:

PRG_SFC.Trans1:

```
Trans1 := (nCount<=100);
```

PRG_SFC:

```
nCount := nCount+1;
IF Trans1 = TRUE THEN // IF nCount<=100 THEN ...
    nVar:=1;
ELSE
    nVar:=2;
END_IF
```

Call of transition Trans1 in an SFC POU:

The screenshot displays the TIA Portal interface. On the left, the 'Solution Explorer' shows a project structure with 'Project10' expanded to 'PRG_SFC (PRG)'. The right pane shows the 'PRG_SFC.ACT1' SFC diagram. The diagram consists of states: 'Init', 'Count', and 'Step1'. Transitions are labeled: 'bVar AND bVar1' (between Init and Count), 'TRUE' (between Count and Step1), and 'Trans1' (between Step1 and Init). A callout box 'ACT1' is connected to the 'Count' state. Two red callout boxes provide context: one points to the 'bVar AND bVar1' transition with the text 'Transition condition entered directly', and another points to the 'Trans1' transition with the text 'Transition programmed in ST'. The code snippets above show the ST logic for 'Trans1 := (nCount <= 1000) AND bVar;' and the SFC program structure.

Access to VAR_IN_OUT variables of the function block in a method/transition/property

In principle, the VAR_IN_OUT variables of a function block can be accessed in a method, transition or property of the function block. Note the following for this type of access:

- If the body or an action of the function block is called from outside the FB, the compiler ensures that the VAR_IN_OUT variables of the function block are assigned with this call.
- This is not the case if a method, transition or property of the function block is called, since the VAR_IN_OUT variables of the FB cannot be assigned within a method, transition or property call. Therefore, access to the VAR_IN_OUT variables might occur by calling the method/transition/property

before the VAR_IN_OUT variables are assigned to a valid reference. Since this would mean invalid access at runtime, accessing the VAR_IN_OUT variables of the FB in a method, transition or property is potentially risky.

Therefore, the following warning with ID C0371 is issued if the VAR_IN_OUT variables of the FB are accessed in a method, transition or property:

„Warning: Access to VAR_IN_OUT <Var> declared in <POU> from external context <Method/Transition/Property>”

An adequate response to this warning could be to check the VAR_IN_OUT variables within the method/transition/property before it is accessed. The operator `__ISVALIDREF` can be used for this check, to ascertain whether a reference refers to a valid value. If this check is enabled, it can be assumed that the user is aware of the risk that potentially exists when the VAR_IN_OUT variables of the FB are accessed in a method/transition/property. Checking the reference is regarded as adequate handling of this risk. The corresponding warning can therefore be suppressed via attribute 'warning disable'.

A sample implementation of a method is shown below.

Function block FB_Sample:

```
FUNCTION_BLOCK FB_Sample
VAR_IN_OUT
    bInOut : BOOL;
END_Var
```

Method FB_Sample.MyMethod:

```
METHOD MyMethod
VAR_INPUT
END_VAR

// The warning can be disabled here as the user is aware of the risk that the reference may not be
// valid by checking its validity
{warning disable C0371}

// Checking the VAR_IN_OUT reference, leave the current method in case of invalid reference
IF NOT __ISVALIDREF(bInOut) THEN
    RETURN;
END_IF


// Access to VAR_IN_OUT reference (only if the reference was confirmed as valid before)
bInOut := NOT bInOut;

// The warning may be restored at the end of the access area
{warning restore C0371}
```

See also:

- [Object-oriented programming \[► 169\]](#)
- [Extending a function block \[► 191\]](#)
- [Reference Programming: SFC elements step and transition \[► 644\]](#)

7.4.4 Object Method

Symbol: 

The object is used for object-oriented programming.

A method contains a sequence of statements. In contrast to a function, it is not an independent POU, but has to be assigned to a function block or a program.

You can use interfaces to organize methods.

Notes on methods

- All data of a method are temporary and are only valid while the method is executed (stack variables). This means that TwinCAT re-initializes all variables and function blocks, which you have declared in a method, with each call of the method.

- Like functions, methods can return a return value.
- According to the IEC 61131-3 standard, methods can have additional inputs and outputs, like normal functions. You assign the inputs and outputs when the method is called.
 - **From TwinCAT 3.1.4026:** Inputs without an explicitly specified initial value must be assigned when the method is called. Inputs with an explicitly specified initial value can be optionally assigned or ignored when the method is called.
- Access to the function block instance or program variables is permitted in the implementation part of a method.
- Use the THIS pointer to point to your own instance.
- It is not possible to access VAR_TEMP variables of the function block in a method.
- You can declare VAR_INST variables for which no reinitialization takes place when methods are called up again (see also [chapter Instance variables](#)).
- By using the return type "REFERENCE TO <structured type>", you can access a single element of the structured data type returned by the method directly when calling the method. For more information see section "[Access to a single element of a structured return type during method/function/property call \[► 94\]](#)".
- In principle, access to VAR_IN_OUT variables of a function block is possible in a method. Since this access is potentially risky, it should be used advisedly. Further information can be found in section "[Access to VAR IN_OUT variables of a function block in a method/transition \[► 95\]](#)".
- Methods that are defined in an interface may only define input, output and VAR_IN_OUT variables, but they may not contain implementations.

Sample:

The code in the following sample causes TwinCAT to write the return value and the outputs of the method to locally declared variables.

Declaration part of "Method1" of the function block FB_Sample:

```
METHOD Method1 : BOOL
VAR_INPUT
  nIn1  : INT;
  bIn2  : BOOL;
END_VAR
VAR_OUTPUT
  fOut1 : REAL;
  sOut2 : STRING;
END_VAR
// <method implementation code>
```

MAIN program:

```
PROGRAM MAIN
VAR
  fbSample      : FB_Sample;
  bReturnValue  : BOOL;
  nLocalInput1 : INT;
  bLocalInput2 : BOOL;
  fLocalOutput1 : REAL;
  sLocalOutput2 : STRING;
END_VAR
bReturnValue := fbSample.Method1(nIn1 := nLocalInput1,
                                bIn2 := bLocalInput2,
                                fOut1 => fLocalOutput1,
                                sOut2 => sLocalOutput2);
```

Creating an object Method

1. Select a function block or a program in the **Solution Explorer** in the PLC project tree.
2. In the context menu select the command **Add > Method...**
 - ⇒ The dialog **Add Method** opens.
3. Enter a name and select a return type, the implementation language, and optionally an access modifier
4. Click on **Open**.

⇒ The object is added to the PLC project tree and opens in the editor. The editor consists of the declaration editor at the top and the implementation part at the bottom.







Dialog Add method

Name	Name of the method The standard methods FB_Init and FB_Exit are offered in a selection list if they are not already inserted below the function block. If it is a derived function block, the selection list also offers all methods of the basic function block.
Return type	Type of the value that is returned
Implementation language	Implementation language selection list

Access modifier

Access modifier	<p>Regulates access to the data</p> <ul style="list-style-type: none"> • PUBLIC: Access is not restricted (equivalent to specifying no access modifier). • PRIVATE: Access to the method is restricted to the function block or the program respectively. • PROTECTED: Access to the method is restricted to the program or the function block and its derivatives respectively. • INTERNAL: Access to the method is limited to the namespace (the library). <p>In addition to these access modifiers, you can manually add the FINAL modifier to a method:</p> <ul style="list-style-type: none"> • FINAL: Overwriting the method in a derivative of the function block is not allowed. This means that the method may not be overwritten/extended in a possibly existing subclass.
Abstract	<p><input checked="" type="checkbox"/> : Indicates that the method has no implementation and that the implementation is provided by the derived FB.</p> <p>Background information on the ABSTRACT keyword can be found under ABSTRACT concept [► 201].</p>

Methods with a different access modifier than PUBLIC are marked with a signal symbol in the Solution Explorer in the PLC project tree.

Access modifier	Object icon	Signal symbol
PRIVATE		 (Lock)
PROTECTED		 (Star)
INTERNAL		 (Heart)



If you copy or move a method from a POU to an interface, TwinCAT automatically deletes the included implementations.

Special methods for a function block

FB_init	<p>Declarations are implicit by default. Explicit declaration are also possible.</p> <p>Contains initialization code for the function block, as defined in the declaration part of the function block.</p> <p>(Methods FB_init, FB_reinit and FB_exit [► 848])</p>
FB_reinit	<p>Explicit declaration required. This is called when the instance of the function block was copied (like during an online change). It re-initializes the new instance module.</p> <p>(Methods FB_init, FB_reinit and FB_exit [► 848])</p>
FB_exit	<p>Explicit declaration required.</p> <p>Call for each instance of the function block before another download or a reset or during an online change for all moved or deleted instances.</p> <p>(Methods FB_init, FB_reinit and FB_exit [► 848])</p>
Properties and interface properties	<p>They each consist of a Set and/or a Get accessor method.</p> <p>(Object Interface property [► 107], Object Property [► 96])</p>

Calling a method

Syntax:

```
<return value variable> := <POU name>.<method name>(<method input name> :=
<variable name> (, <further method input name> := <variable name> )* );
```

When you call the method, you assign transfer parameters to the input variables of the method. In doing so, observe the declaration. It is sufficient to specify the names of the input variables without considering their order in the declaration.

Sample:

The code in the following sample causes TwinCAT to write the return value and the outputs of the method to locally declared variables.

Declaration part of "Method1" of the function block FB_Sample:

```
METHOD Method1 : BOOL
VAR_INPUT
    nIn1 : INT;
    bIn2 : BOOL;
END_VAR
VAR_OUTPUT
    fOut1 : REAL;
    sOut2 : STRING;
END_VAR
// <method implementation code>
```

MAIN program:

```
PROGRAM MAIN
VAR
    fbSample : FB_Sample;
    bReturnValue : BOOL;
    nLocalInput1 : INT;
    bLocalInput2 : BOOL;
    fLocalOutput1 : REAL;
    sLocalOutput2 : STRING;
END_VAR
bReturnValue := fbSample.Method1(nIn1 := nLocalInput1,
                                bIn2 := bLocalInput2,
                                fOut1 => fLocalOutput1,
                                sOut2 => sLocalOutput2);
```

Access to a single element of a structured return type during method/function/property call

The following implementation can be used to directly access an individual element of the structured data type that is returned by the method/function/property when a method, function or property is called. A structured data type is, for example, a structure or a function block.

1. The return type of the method/function/property is defined as "REFERENCE TO <structured type>" (instead of just "<structured type>").
2. Note that with such a return type – if, for example, an FB-local instance of the structured data type is to be returned – the reference operator REF= must be used instead of the "normal" assignment operator :=.

The declarations and the sample in this section refer to the call of a property. However, they are equally transferrable to other calls that deliver return values (e.g. methods or functions).

Sample

Declaration of the structure ST_Sample (structured data type):

```
TYPE ST_Sample :
STRUCT
    bVar : BOOL;
    nVar : INT;
END_STRUCT
END_TYPE
```

Declaration of the function block FB_Sample:

```
FUNCTION_BLOCK FB_Sample
VAR
    stLocal      : ST_Sample;
END_VAR
```

Declaration of the property FB_Sample.MyProp with the return type "REFERENCE TO ST_Sample":

```
PROPERTY MyProp : REFERENCE TO ST_Sample
```

Implementation of the Get method of the property FB_Sample.MyProp:

```
MyProp REF= stLocal;
```

Implementation of the Set method of the property FB_Sample.MyProp:

```
stLocal := MyProp;
```

Calling the Get and Set methods in the main program MAIN:

```
PROGRAM MAIN
VAR
    fbSample      : FB_Sample;
    nSingleGet    : INT;
    stGet         : ST_Sample;
    bSet          : BOOL;
    stSet         : ST_Sample;
END_VAR

// Get - single member and complete structure possible
nSingleGet := fbSample.MyProp.nVar;
stGet      := fbSample.MyProp;

// Set - only complete structure possible
IF bSet THEN
    fbSample.MyProp REF= stSet;
    bSet              := FALSE;
END_IF
```

Through the declaration of the return type of the property MyProp as "REFERENCE TO ST_Sample" and through the use of the reference operator REF= in the Get method of this property, a single element of the returned structured data type can be accessed directly on calling the property.

```
VAR
    fbSample      : FB_Sample;
    nSingleGet    : INT;
END_VAR

nSingleGet := fbSample.MyProp.nVar;
```

If the return type were only to be declared as "ST_Sample", the structure returned by the property would first have to be assigned to a local structure instance. The individual structure elements could then be queried on the basis of the local structure instance.

```
VAR
    fbSample      : FB_Sample;
    stGet         : ST_Sample;
    nSingleGet    : INT;
END_VAR

stGet      := fbSample.MyProp;
nSingleGet := stGet.nVar;
```

Access to VAR_IN_OUT variables of the function block in a method/transition/property

In principle, the VAR_IN_OUT variables of a function block can be accessed in a method, transition or property of the function block. Note the following for this type of access:

- If the body or an action of the function block is called from outside the FB, the compiler ensures that the VAR_IN_OUT variables of the function block are assigned with this call.
- This is not the case if a method, transition or property of the function block is called, since the VAR_IN_OUT variables of the FB cannot be assigned within a method, transition or property call. Therefore, access to the VAR_IN_OUT variables might occur by calling the method/transition/property before the VAR_IN_OUT variables are assigned to a valid reference. Since this would mean invalid access at runtime, accessing the VAR_IN_OUT variables of the FB in a method, transition or property is potentially risky.

Therefore, the following warning with ID C0371 is issued if the VAR_IN_OUT variables of the FB are accessed in a method, transition or property:

„Warning: Access to VAR_IN_OUT <Var> declared in <POU> from external context <Method/Transition/Property>”

An adequate response to this warning could be to check the VAR_IN_OUT variables within the method/transition/property before it is accessed. The operator `__ISVALIDREF` can be used for this check, to ascertain whether a reference refers to a valid value. If this check is enabled, it can be assumed that the user is aware of the risk that potentially exists when the VAR_IN_OUT variables of the FB are accessed in a method/transition/property. Checking the reference is regarded as adequate handling of this risk. The corresponding warning can therefore be suppressed via attribute 'warning disable'.

A sample implementation of a method is shown below.

Function block FB_Sample:

```
FUNCTION_BLOCK FB_Sample
VAR_IN_OUT
    bInOut : BOOL;
END_Var
```

Methode FB_Sample.MyMethod:

```
METHOD MyMethod
VAR_INPUT
END_VAR

// The warning can be disabled here as the user is aware of the risk that the reference may not be
// valid by checking its validity
{warning disable C0371}

// Checking the VAR_IN_OUT reference, leave the current method in case of invalid reference
IF NOT __ISVALIDREF(bInOut) THEN
    RETURN;
END_IF


// Access to VAR_IN_OUT reference (only if the reference was confirmed as valid before)
bInOut := NOT bInOut;

// The warning may be restored at the end of the access area
{warning restore C0371}
```

See also:

- [Object-oriented programming \[► 169\]](#)
- [Object Interface \[► 102\]](#)
- [Implementation of an interface \[► 198\]](#)
- [Extending a function block \[► 191\]](#)
- [Method call \[► 199\]](#)
- [ABSTRACT concept \[► 201\]](#)
- [Reference Programming > Instance Variables - VAR_INST \[► 688\]](#)

7.4.5 Object Property

Symbol: 

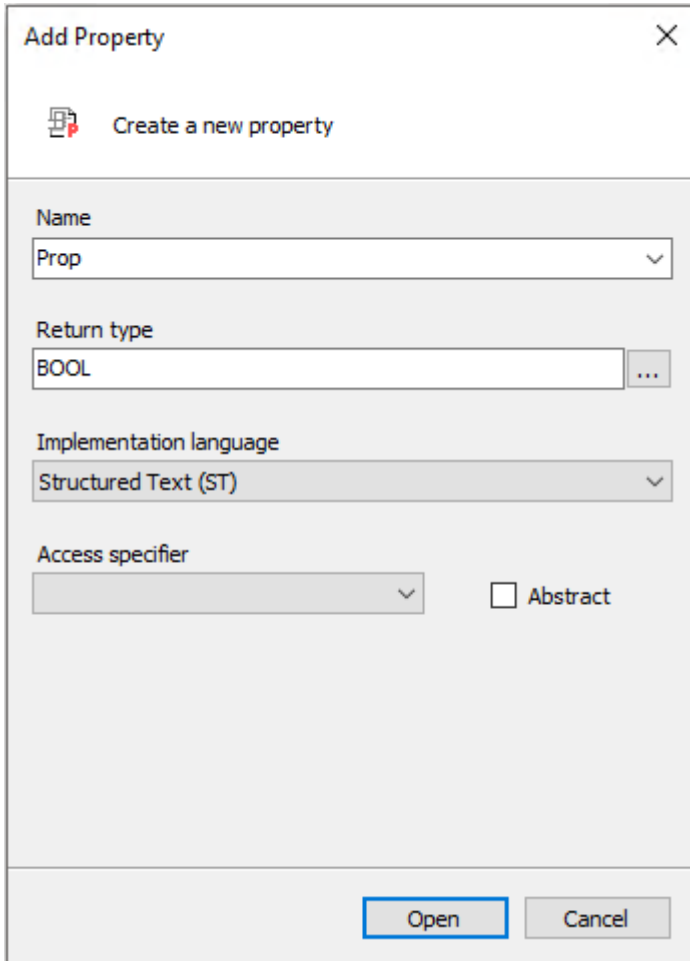
A property is an extension of the IEC 61131-3 standard and is a means for object-oriented programming. It consists of the accessor methods Get and Set. TwinCAT automatically calls these methods when a read or write access occurs to the function block that implements the property.

Object Creating a property

1. Select a function block or a program in the **Solution Explorer** in the PLC project tree.
2. In the context menu select the command **Add > Property...**

- ⇒ The **Add Property** dialog opens.
- 3. Enter a name and select a return type, the implementation language, and optionally an access modifier.
- 4. Click on **Open**.
- ⇒ The object is added to the PLC project tree and opens in the editor.

Dialog Add property



Name	Name of the property
Return type	Type of the value that will be returned (default type or structured type)
Implementation language	Implementation language selection list

Access modifier

Access specifier	<p>Regulates access to the data</p> <ul style="list-style-type: none"> • PUBLIC: Access is not restricted (equivalent to specifying no access modifier). • PRIVATE: Access to the property is restricted to the function block or the program respectively. • PROTECTED: Access to the property is restricted to the program or the function block and its derivatives respectively. • INTERNAL: Access to the property is limited to the namespace (the library). <p>In addition to these access modifiers, you can manually add the FINAL modifier to a property:</p> <p>FINAL: Overwriting the property in a derivative of the function block is not allowed. This means that the property may not be overwritten/extended in a possibly existing subclass.</p>
Abstract	<p><input checked="" type="checkbox"/> : Indicates that the property has no implementation and that the implementation is provided by the derived FB.</p> <p>Background information on the ABSTRACT keyword can be found under ABSTRACT concept [► 201].</p>

Properties with a different access modifier than PUBLIC are marked with a signal symbol in the Solution Explorer in the PLC project tree.

Access modifier	Object icon	Signal symbol
PRIVATE		(Lock)
PROTECTED		(Star)
INTERNAL		(Heart)

In addition, a property can contain local variables. However, a property cannot contain additional inputs. In contrast to a function or method, it also cannot contain additional outputs.



If you copy or move a property from a POU to an interface, TwinCAT automatically deletes the included implementations.

Get and Set accessors

TwinCAT automatically adds the Get and Set accessor methods below the property object in the PLC project tree. The **Add** command can be used to add them explicitly.

TwinCAT calls the Set accessor when write access to the property occurs, i.e. when you use the property name as input parameter.

TwinCAT calls the Get accessor when read access to the property occurs, i.e. when you use the property name as output parameter.

Please note that you have to implement the accessor methods in order to access the property.

Implementation sample

Declaration of the function block FB_Sample

```
FUNCTION_BLOCK FB_Sample
VAR
  nVar : INT;
END_VAR
```

```
nVar := nVar + 1;
```

Declaration of the property nValue

```
PROPERTY PUBLIC nValue : INT
```

Implementation of the accessor method FB_Sample.nValue.Set

```
nVar := nValue;
```

Implementation of the accessor method FB_Sample.nValue.Get

```
nValue := nVar;
```

Calling the property nValue

```
PROGRAM MAIN
VAR
    fbSample : FB_Sample;
END_VAR

fbSample();
If fbSample.nValue > 500 THEN
    fbSample.nValue := 0;
END_IF;
```

If you only want to use the property for read access or only for write access, you can delete the unused accessor.



You can add access specifiers to the accessor methods in the following places:

- Through manual entries in the declaration part of the accessor.
- In the **Add 'get' accessor** or **Add 'set' accessor** dialog, when you add the accessor explicitly with the **Add** command.



Compatibility warning for properties of type REFERENCE

In TC3.1 Build 4022 the calling behavior of properties defined with the return type 'REFERENCE TO <...>' changes.

For write accesses using ':=' , with versions < 3.1.4022.0 the set accessor is called so that the reference is written. With versions >= 3.1.4022.0, however, the get-accessor is called so that the value is written. To assign the reference with versions >= 3.1.4022.0, the reference assignment operator 'REF= [[▶ 629](#)]' must be used.

Monitoring for properties in online mode

Pragmas are available for monitoring for properties in online mode, which should be added at the top of the definition property ([Attribute 'monitoring'](#) [[▶ 810](#)]):

- {attribute 'monitoring':='variable'}: With each access to the property, TwinCAT stores the actual value in a variable and displays the value of this variable. This value may become obsolete, if the code no longer accesses the property.
- {attribute 'monitoring' := 'call'}: Each time the value is displayed, TwinCAT calls the code of the Get accessor. If this code contains a side effect, the side effect is executed by the monitoring.

You can monitor a property using the following features:

- Inline-Monitoring
Requirement: In the TwinCAT options in the category **TwinCAT > PLC Programming Environment > Text Editor** the option **Enable Inline-Monitoring** is activated in the tab **Monitoring** .
- Watch List ([Using Watchlists](#) [[▶ 226](#)])

Access to a single element of a structured return type during method/function/property call

The following implementation can be used to directly access an individual element of the structured data type that is returned by the method/function/property when a method, function or property is called. A structured data type is, for example, a structure or a function block.

1. The return type of the method/function/property is defined as "REFERENCE TO <structured type>" (instead of just "<structured type>").

- Note that with such a return type – if, for example, an FB-local instance of the structured data type is to be returned – the reference operator REF= must be used instead of the "normal" assignment operator :=.

The declarations and the sample in this section refer to the call of a property. However, they are equally transferrable to other calls that deliver return values (e.g. methods or functions).

Sample

Declaration of the structure ST_Sample (structured data type):

```
TYPE ST_Sample :
STRUCT
  bVar : BOOL;
  nVar : INT;
END_STRUCT
END_TYPE
```

Declaration of the function block FB_Sample:

```
FUNCTION_BLOCK FB_Sample
VAR
  stLocal : ST_Sample;
END_VAR
```

Declaration of the property FB_Sample.MyProp with the return type "REFERENCE TO ST_Sample":

```
PROPERTY MyProp : REFERENCE TO ST_Sample
```

Implementation of the Get method of the property FB_Sample.MyProp:

```
MyProp REF= stLocal;
```

Implementation of the Set method of the property FB_Sample.MyProp:

```
stLocal := MyProp;
```

Calling the Get and Set methods in the main program MAIN:

```
PROGRAM MAIN
VAR
  fbSample : FB_Sample;
  nSingleGet : INT;
  stGet : ST_Sample;
  bSet : BOOL;
  stSet : ST_Sample;
END_VAR

// Get - single member and complete structure possible
nSingleGet := fbSample.MyProp.nVar;
stGet := fbSample.MyProp;

// Set - only complete structure possible
IF bSet THEN
  fbSample.MyProp REF= stSet;
  bSet := FALSE;
END_IF
```

Through the declaration of the return type of the property MyProp as "REFERENCE TO ST_Sample" and through the use of the reference operator REF= in the Get method of this property, a single element of the returned structured data type can be accessed directly on calling the property.

```
VAR
  fbSample : FB_Sample;
  nSingleGet : INT;
END_VAR

nSingleGet := fbSample.MyProp.nVar;
```

If the return type were only to be declared as "ST_Sample", the structure returned by the property would first have to be assigned to a local structure instance. The individual structure elements could then be queried on the basis of the local structure instance.

```
VAR
  fbSample : FB_Sample;
  stGet : ST_Sample;
  nSingleGet : INT;
END_VAR
```



```
stGet      := fbSample.MyProp;
nSingleGet := stGet.nVar;
```

Access to VAR_IN_OUT variables of the function block in a method/transition/property

In principle, the VAR_IN_OUT variables of a function block can be accessed in a method, transition or property of the function block. Note the following for this type of access:

- If the body or an action of the function block is called from outside the FB, the compiler ensures that the VAR_IN_OUT variables of the function block are assigned with this call.
- This is not the case if a method, transition or property of the function block is called, since the VAR_IN_OUT variables of the FB cannot be assigned within a method, transition or property call. Therefore, access to the VAR_IN_OUT variables might occur by calling the method/transition/property before the VAR_IN_OUT variables are assigned to a valid reference. Since this would mean invalid access at runtime, accessing the VAR_IN_OUT variables of the FB in a method, transition or property is potentially risky.

Therefore, the following warning with ID C0371 is issued if the VAR_IN_OUT variables of the FB are accessed in a method, transition or property:

„Warning: Access to VAR_IN_OUT <Var> declared in <POU> from external context <Method/Transition/Property>”

An adequate response to this warning could be to check the VAR_IN_OUT variables within the method/transition/property before it is accessed. The operator `__ISVALIDREF` can be used for this check, to ascertain whether a reference refers to a valid value. If this check is enabled, it can be assumed that the user is aware of the risk that potentially exists when the VAR_IN_OUT variables of the FB are accessed in a method/transition/property. Checking the reference is regarded as adequate handling of this risk. The corresponding warning can therefore be suppressed via attribute 'warning disable'.

A sample implementation of a method is shown below.

Function block FB_Sample:

```
FUNCTION_BLOCK FB_Sample
VAR_IN_OUT
    bInOut : BOOL;
END_Var
```

Methode FB_Sample.MyMethod:

```
METHOD MyMethod
VAR_INPUT
END_VAR

// The warning can be disabled here as the user is aware of the risk that the reference may not be
// valid by checking its validity
{warning disable C0371}

// Checking the VAR_IN_OUT reference, leave the current method in case of invalid reference
IF NOT __ISVALIDREF(bInOut) THEN
    RETURN;
END_IF

// Access to VAR_IN_OUT reference (only if the reference was confirmed as valid before)
bInOut := NOT bInOut;

// The warning may be restored at the end of the access area
{warning restore C0371}
```

Initialization example:

In the following sample, an input variable and a property are initialized by a function block that has an [FB_init](#) method [► 850] with an additional parameter.

Function block FB_Sample:

```
FUNCTION_BLOCK FB_Sample
VAR_INPUT
    nInput      : INT;
END_VAR
VAR
```

```

    nLocalInitParam : INT;
    nLocalProp      : INT;
END_VAR

```

Method FB_Sample.FB_init:

```

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)
    nInitParam   : INT;
END_VAR
nLocalInitParam := nInitParam;

```

Property FB_Sample.nMyProperty and the associated Set function:

```

PROPERTY nMyProperty : INT
nLocalProp := nMyProperty;

```

Program MAIN:

```

PROGRAM MAIN
VAR
    fbSample : FB_Sample(nInitParam := 1) := (nInput := 2, nMyProperty := 3);
    aSample  : ARRAY[1..2] OF FB_Sample[(nInitParam := 4), (nInitParam := 7)]
              := [(nInput := 5, nMyProperty := 6), (nInput := 8, nMyProperty := 9)];
END_VAR

```


Initialization result:

- fbSample
 - nInput = 2
 - nLocalInitParam = 1
 - nLocalProp = 3
- aSample[1]
 - nInput = 5
 - nLocalInitParam = 4
 - nLocalProp = 6
- aSample[2]
 - nInput = 8
 - nLocalInitParam = 7
 - nLocalProp = 9

See also:

- [Object Interface property \[► 107\]](#)
- [Object-oriented programming \[► 169\]](#)
- [Extending a function block \[► 191\]](#)
- [Reference programming > Methods FB_init, FB_reinit and FB_exit \[► 848\]](#)

7.4.6 Object Interface

Symbol: 

Keyword: INTERFACE

An interface is a tool for object-oriented programming. The object **Interface** describes a set of method and property prototypes. Prototype in this context means that the methods and properties only contain declarations, but no implementation.

In this way you can use various function blocks with common properties in the same way.

You can add the objects **Interface property** and **Interface method** to the **Interface** object.



Creating an object Interface

1. Select a folder in the **Solution Explorer** in the PLC project tree.
2. In the context menu select the command **Add > Interface...**
 - ⇒ The **Add Interface** dialog opens.
3. Enter a name, and optionally select an interface to be extended.
4. Click on **Open**.
 - ⇒ The interface is added to the PLC project tree and opens in the editor.

Dialog Add Interface

Name	Interface name
------	----------------

Inheritance

Advanced	 : Extends the interface that is entered in the input field or selected via the input assistant  . This means that all interface methods, which the new interface extends, are also available in the new interface.
----------	---

Interface applications

1. Checking the interface declaration via the compiler

- In an interface (e.g. I_Sample) you declare the methods and properties (including return type, inputs etc.), which are to be associated with this interface.
- In the function blocks, which are to correspond to this interface and should therefore make the corresponding methods and properties available, you implement the interface I_Sample.
 - The function block contains the interface in the IMPLEMENTS list within its declaration part (e.g. FB_Sample IMPLEMENTS I_Sample).
 - A function block can implement one or several interfaces (e.g. FB_Sample IMPLEMENTS I_Sample1, I_Sample2).
- A function block, which implements an interface, must contain all methods and properties that are defined in this interface (interface methods and properties). The declaration of the methods and properties must match the declaration in the interface exactly (name, return type, inputs, outputs).
- The function blocks add function block-specific code to the interface methods and interface properties. If an interface is implemented by several function blocks, you can use the same method with the parameters but different implementation code in different function blocks.
- For function blocks, which implement one or several interfaces, the compiler checks whether the function blocks meet the respective interface declarations. If the element declarations in the interface and in the function block differ, or if the interface contains further elements, which are not included in the function block, the compiler reports an error.

2. Calling methods and properties of a function block instance via an interface variable

In addition to automatic verification of the interface declaration via the compiler, you can use interfaces to call an interface method or interface property of a function block instance via an interface variable.

- First, instantiate the interface (e.g. iSample : I_Sample;) and the function block(s), which implement(s) the interface correctly (see use case 1: Checking the interface declaration via the compiler).
- You can then assign the interface variable to each instance of a function block, which implements the interface correctly. If an interface variable has not yet been assigned, the variable contains the value 0 in online mode.
- In the last step, you can call an interface method or property via the interface variable. The method or property is called for the function block, to which the interface refers.

- Such an implementation enables the use of different, but similar function blocks via the interface variable in a consistent manner. Depending on the project state you can assign a particular function block instance to the interface variable, for example, so that the call of the interface methods and properties is identical, although a different function block instance is used, depending on the project state.



An interface type variable has to be assigned the instance of a function block, before a method or property can be called via the interface variable.

Notes

- You cannot declare variables within an interface. An interface has no implementation part and no actions. Only a collection of methods and properties is defined, which contain declaration code, but no implementation code.
- An interface type variable is a reference to instances of function blocks. TwinCAT always treats variables, which are defined as an interface type, as references.

Interface references and Online Change

- Interface references are automatically redirected from TwinCAT 3.0 build 3100, so that the correct interface is always referenced, even in the event of an online change. This requires additional code and time, which may cause issues, depending on the number of affected objects. Before the online change is performed, programmers are therefore shown the number of affected variables and interface references, so that they can decide whether to go ahead with the online change or abort it.

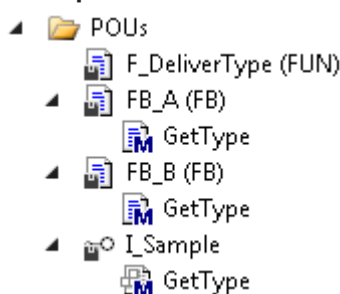
Checking interface variables

- The operator `__ISVALIDREF` can only be used for operands of type `REFERENCE TO`. This operator cannot be used for checking interface variables. To check whether an interface variable was already assigned a function block instance, you can check the interface variable for not equal to 0 (`IF iSample <> 0 THEN ...`).

Extended monitoring of an interface variable

- Advanced options for monitoring/debugging an interface variable are available from TwinCAT 3.1 Build 4024. An interface variable in the monitoring area (declaration editor, watch list) can be expanded. The symbol path and online data of the currently assigned FB instance are then displayed below the interface variable.

Example 1



Interface declaration:

- You have added the interface `I_Sample` to your project. Add the method `GetType` with the return type `STRING` to the interface.
- `I_Sample` and `GetType` contain no implementation code. The method `GetType` contains only the required (variable) declarations (e.g. return type). You can program out the method `GetType` later on in the function block that implements the interface `I_Sample`.

```
INTERFACE I_Sample
```

Method `I_Sample.GetType`:

```
METHOD GetType : STRING
```

Interface implementation:

- If you subsequently add a function block to the project and enter the interface I_Sample in the field **Implements** in the dialog **Add**, TwinCAT also automatically adds the method GetType to this function block. Here you can implement function block-specific code in the methods.
- The function blocks FB_A and FB_B both implement the interface I_Sample:

```
FUNCTION_BLOCK FB_A IMPLEMENTS I_Sample
```

```
FUNCTION_BLOCK FB_B IMPLEMENTS I_Sample
```

- Both function blocks therefore have to include a method with the name GetType and the return type STRING. Otherwise the compiler reports an error (see use case 1 in section [Interface application \[► 103\]s](#)).

Method FB_A.GetType:

```
METHOD GetType : STRING
```

```
GetType := 'FB_A';
```

Method FB_B.GetType:

```
METHOD GetType : STRING
```

```
GetType := 'FB_B';
```

Interface application:

- A function F_DeliverType contains the declaration of an input variable of the type of interface I_Sample. Within the function, the interface method GetType is called via the interface variable iSample. In this case, whether FB_A.GetType or FB_B.GetType is called depends on the transferred function block type (see application 2 in section [Interface application \[► 103\]s](#)).

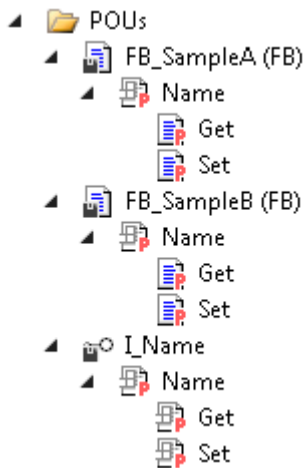
```
FUNCTION F_DeliverType : STRING
VAR_INPUT
    iSample : I_Sample;
END_VAR

F_DeliverType := iSample.GetType();
```

- Instances of function blocks, which implement the interface I_Sample (e.g. FB_A and FB_B), can be assigned to the input variable of the function F_DeliverType.
- Examples of function calls:
 - If the function block instance fbA is transferred to the function F_DeliverType, the method fbA.GetType will be called inside the function since the interface variable iSample points to the function block instance fbA. This method call delivers the return value 'FB_A', which in turn is returned by the function F_DeliverType and assigned in the main program to the variable sResultA.
 - Accordingly, sResultB receives the value 'FB_B', since the method fbB.GetType is called inside the function F_DeliverType.

```
PROGRAM MAIN
VAR
    fbA      : FB_A;
    fbB      : FB_B;
    sResultA : STRING;
    sResultB : STRING;
END_VAR

sResultA := F_DeliverType(iSample := fbA); // call with instance of type FB_A
sResultB := F_DeliverType(iSample := fbB); // call with instance of type FB_B
```

Example 2**Interface declaration:**

- You have added the interface `I_Name` to your project. Add the property `Name` with the return type `STRING` to the interface. The property has the accessor methods `Get` and `Set`. The `Get` accessor can be used to read the name of any object from a function block that implements the interface. The `Set` accessor is used to write the name into this function block.
- `I_Name` and `Name` contain no implementation code. The property `Name` only contains the required declaration (return type). Therefore you cannot process the `Get` and `Set` methods inside the interface definition, but you can do this later in the function block that implements the interface `I_Name`.

```
INTERFACE I_Name
```

Property `I_Name.Name`:

```
PROPERTY Name : STRING
```

Interface implementation:

- The function blocks `FB_SampleA` and `FB_SampleB` implement the interface `I_Name`.
- If the interface is specified, for example, when creating the function blocks in the dialog **Add**, TwinCAT automatically adds the property `Name` with the `Get` and `Set` methods under the function blocks `FB_SampleA` and `FB_SampleB`.
- You can edit the accessor methods underneath the function blocks, for example so that the variable `sVar1` is read and you thus obtain the name of an object. In `FB_SampleB`, which implements the same interface `I_Name`, you can implement the `Get` method code, which then returns the name of another object. The `Set` method can be used to write the name, which the MAIN program supplies ('abc'), into the function block `FB_SampleB`.
- The function blocks `FB_SampleA` and `FB_SampleB` each implement the interface `I_Name`:

```
FUNCTION_BLOCK FB_SampleA IMPLEMENTS I_Name
VAR
    sVar1 : STRING := 'My name is A.';
END_VAR

FUNCTION_BLOCK FB_SampleB IMPLEMENTS I_Name
VAR
    sVar2 : STRING := 'My name is B.';
END_VAR
```

- Both function blocks must therefore contain a property with the name `Name` and the return type `STRING`. The property must have a `Get` and a `Set` method. Otherwise the compiler reports an error (see use case 1 in section [Interface applications \[▶ 103\]](#)).

FB_SampleA.Name.Get:

```
Name := sVar1;
```

FB_SampleA.Name.Set:

```
sVar1 := Name;
```

FB_SampleB.Name.Get:

```
Name := sVar2;
```

FB_SampleB.Name.Set:

```
sVar2 := Name;
```

Interface application:

- The properties of the function blocks can be accessed both via the corresponding function block instances and via an interface variable of the type `I_Name`. The prerequisite for access via an interface variable is that this variable has been assigned a specific function block instance beforehand that implements the interface `I_Name`.

```
PROGRAM MAIN
VAR
    iName      : I_Name;

    fbSampleA : FB_SampleA;
    sNameA    : STRING;      // will be 'My name is A.'

    fbSampleB : FB_SampleB;
    sNameB    : STRING;      // will be 'My name is B.' after first cycle
                                // and will be 'New name' afterwards
END_VAR

// assign FB instance fbSample1 to interface variable
iName := fbSampleA;


// access to name property of fbSample1 via interface variable (Get)
sNameA := iName.Name;

// access to name property of fbSample2 via FB instance (Get and Set)
sNameB := fbSampleB.Name;
fbSampleB.Name := 'New name';
```

See also:

- [Implementing an interface \[► 198\]](#)
- [Extending an interface \[► 197\]](#)

7.4.6.1 Object Interface method

Symbol: 

An interface method is a means of object-oriented programming. The **Interface method** object can be added to an interface via the command **Add > Method...**


If a method is added under an interface, you can only add and instantiate variable declarations (input, output and input/output variables) in this method.

Program code can only be added to the method, once a function block "implements" the interface that belongs to the method. TwinCAT then adds the method under the function block.

See also:

- [Object Interface \[► 102\]](#)
- [Object Method \[► 90\]](#)
- [Implementation of an interface \[► 198\]](#)

7.4.6.2 Object Interface property

Symbol: 

An interface property is a tool for object-oriented programming. The object **Interface property** is added to an interface via the command **Add > Property...** in order to extend the description of the interface with the accessor method Get and/or Set. No implementation code is included for the accessor methods in the interface property. If you delete the Set accessor, only read access is available for the property, not write access.

The Get accessor is used for read access to the property.
The Set accessor is used for write access to the property.

If the property has no Get and/or Set, this accessor can be added to the interface property with the command **Add**.



If you extend a function block or a program with an interface that contains properties, TwinCAT automatically adds this and the corresponding Get and/or Set accessors in the PLC project tree under the POU. You can then implement the code in the Get and/or Set accessors.

See also:

- [Object Interface \[► 102\]](#)
- [Object Property \[► 96\]](#)
- [Implementation of an interface \[► 198\]](#)

7.5 Creating source code in IEC

Source code

“Source code” indicates the implementation code, which you add to the programming blocks with the aid of the corresponding programming language editors.

Programming language

When you create a programming block, you can decide which implementation language you want to use for the programming. In addition to the IEC languages, CFC is also available.

Programming language editors

A programming block can be opened for editing in the corresponding programming language editor by double-clicking the object in the PLC project tree. The function block therefore appears either in the text-based ST Editor or in one of the graphical editors for FBD/LD/IL, SFC or CFC. Each editor consists of two windows: The upper window is used for the declarations, either in text-based or tabular form, depending on the setting. The lower window is used for the implementation code. The display and the behavior of each editor can be configured for the entire project in the corresponding tab of the TwinCAT options.

Note the option to open a programming block in offline in the editor, even when the application is in online mode. ([Command Edit object \(offline\) \[► 886\]](#))

See also:

- Reference Programming > [Programming languages and their editors \[► 622\]](#)

7.5.1 FBD/LD/IL

A combined editor enables programming in the languages FBD (Function Block Diagram), LD (Ladder Diagram) and IL (Instruction List).

The basic unit of FBD and LD programming is a network. Each network contains a structure, which can represent the following: A logical or arithmetic expression, the call of a programming block (function, function block, program etc.), a jump, or a return statement. IL does not actually require networks. However, in TwinCAT an IL program also consists of at least one network, to support conversion to FBD or LD. An IL programs should therefore also be structured with networks in mind.

See also:

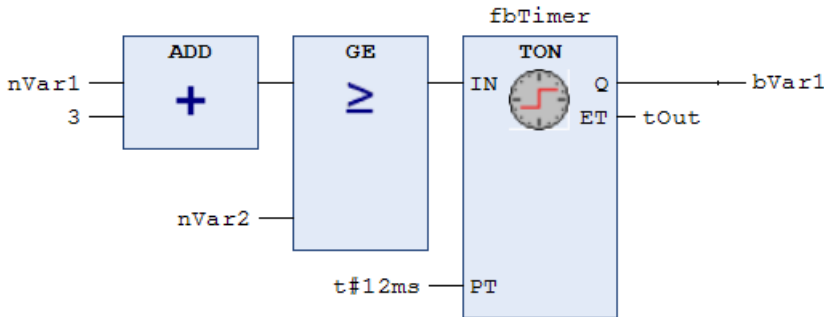
- Reference Programming: [Function Block Diagram / Ladder / Instruction List \(FBD/LD/IL\)](#) [▶ 651]
- TC3 User Interface documentation: [FBD/LD/IL](#) [▶ 1025]

Function Block Diagram (FBD)

Function Block Diagram is a graphical IEC 61131 programming language. It uses a list of networks. Each network contains a structure, which may contain logical and arithmetic expressions, function block calls, a jump, or a return statement.

The function blocks that are used are familiar from Boolean algebra. Function blocks and variables are linked with connecting lines. The signal flow in the network is from left to right. The signal flow in the editor is from top to bottom, starting with network 1.

Example:



Ladder Diagram (LD)

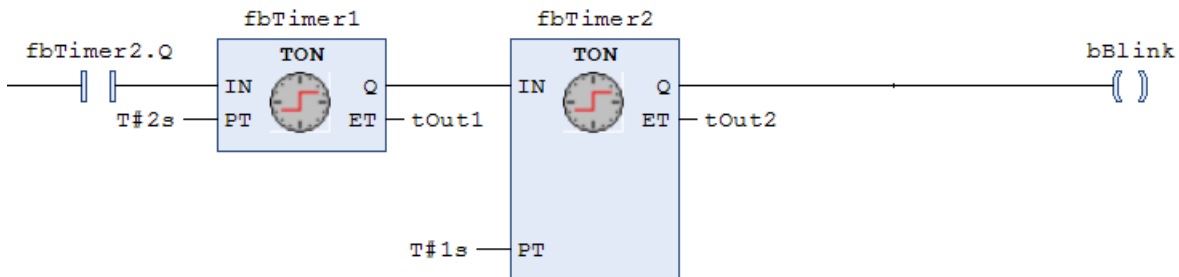
Ladder Diagram (LD) is a graphical programming language, which is based on the principle of electrical circuit diagrams.

Ladder Diagram is suitable for configuring logical control circuits, and it can also be used for creating networks, similar to FBD. LD is therefore well suited for controlling calls of other program blocks.

Ladder Diagram consists of a series of networks. A network is limited on the left by a vertical line (busbar). A network contains a circuit diagram consisting of contacts, coils, optional function blocks (POUs) and connecting lines. On the left side of the network there is a contact or a series of contacts, which pass on the state ON or OFF from left to right, corresponding to the Boolean values TRUE and FALSE. Each contact is associated with a Boolean variable. If this variable is TRUE, the state is passed on from left to right via the connecting line. Otherwise OFF is passed on. In this way, the coils in the right part of the network are assigned the value ON or OFF from the left part. Accordingly, the value TRUE or FALSE is written to in the Boolean variables assigned to them.

If the elements are connected in series, this corresponds to a logical AND link. If they are connected in parallel, it corresponds to a logical OR link. A line through an element indicates negation of the element. Negation of an input or output is indicated by a circle symbol.

Example:



IEC 61131-3 defines a full LD instruction set, consisting of various types of contacts and coils. Contacts conduct current from left to right (according to their type). Coils store the incoming value. Contacts and coils are assigned Boolean variables. They can complement an LD network through jumps, backward jumps, markers and comments.

Instruction List (IL)

Instruction List is an IEC 61131-compliant programming language that is similar to Assembler. It supports accumulator-based programming.

An Instruction List (IL) consists of a series of statements. Each instruction starts in a new row and contains an operator and one or several comma-separated operands, depending on the type of operation. An instruction may be preceded by a marker, followed by a colon. The marker is used to identify the instruction and can be used as jump destination. A comment has to be the last element in a line. Empty lines can be added between instructions.

All IEC 61131-3 operators are supported, as well as multiple inputs, multiple outputs, negations, comments, setting/resetting of outputs and conditional/unconditional jumps.

Each instruction is primarily based on the loading of values into the accumulator (LD instruction). The corresponding operation is then executed with the parameter from the accumulator. The result of the operation is written back into the accumulator, from where it should be stored in a specific location via an ST instruction.

Instruction List supports comparison operators (EQ, GT, LT, GE, LE, NE) and jumps for programming conditional execution or loops. Jumps can be unconditional (JMP) or conditional (JMPC / JMPCN). For conditional jumps the program checks whether the value in the accumulator is TRUE or FALSE.

Example:

1	PROGRAM IL		
2	VAR		
3	fbTimer1	: TON;	
4	fbTimer2	: TON;	
5	bVar	: BOOL;	
6	nVar	: INT;	
7	tIn1	: TIME;	
8	tOut1	: TIME;	
9	END_VAR		

1	LD	bVar	
	ST	fbTimer1.IN	<i>starts timer with rising edge, resets time...</i>
	JMP	mark1	
	CAL	fbTimer1 (
		PT:=tIn1,	
		ET:=>tOut1)	
	LD	fbTimer1.Q	<i>gets TRUE, delay time (PT) after a rising...</i>
	ST	fbTimer2.IN	<i>starts timer with rising edge, resets time...</i>
2	mark1:		
	LD	nVar	
	ADD	230	

7.5.1.1 Programming Function Block Diagrams (FBD)

Creating a POU in the Function Block Diagram implementation language (FBD)

1. Select a folder in the **Solution Explorer** in the PLC project tree.
2. In the context menu select the command **Add > POU...**
 - ⇒ The **Add POU** dialog opens.
3. Enter a name and select the implementation language "Function Block Diagram (FBD)".
4. Click on **Open**.
 - ⇒ The POU is added to the PLC project tree and opened in the editor. It consists of the declaration editor in the upper part and the implementation part with an empty network in the lower part. The **Toolbox** view opens automatically, providing suitable elements, operators and function blocks for FBD programming.

Programming a network

1. Click in the automatically added empty network in the implementation part.
 - ⇒ The network is highlighted in yellow, the part with the network number on the left edge is highlighted in red.
2. Right-click to open the context menu.
 - ⇒ It provides add commands for elements that can be added at this point.
3. Add the required programming elements via the menu commands or by dragging the elements from the **Toolbox** view.
4. Select the command **Insert Assignment**, for example.
 - ⇒ An assignment line is added. Three question marks represent the assignment source and the assignment target.
5. Select the question marks and replace them with the required variable. The input assistant is available.
6. Move the cursor over the assignment line.
 - ⇒ The possible insert positions for further elements are shown as grey hash characters. Click on a hash character to select the position. Once again, the suitable insert instructions become available.
7. Alternatively, you can use the mouse to drag an element from the **Toolbox** view into the network. For example, in the **Toolbox** view click on the function block element, keep the mouse button pressed and move the cursor over the network.
 - ⇒ All possible insert positions are shown in green.
8. Release the mouse button to insert the function block.
 - ⇒ The function block is shown in the network. The internal function block type and the instance name above the box, which is required if a function block is used, are still indicated by three question marks.
9. Select the three question marks within the box and replace them with the function block name. The input assistant is available.
 - ⇒ The inputs and outputs of the selected function block are shown. They are still indicated by question marks, for function blocks also the instance name.

See also:

- TC3 User Interface documentation: [Command Insert Assignment \[► 1026\]](#)

Programming branches (subnetworks)

1. Add a new network in the implementation part of your POU via the command **Insert Network** in the **FBD/LD/IL** menu or from the context menu or from the **Toolbox** view.
2. For example, drag an ADD operator into the empty network and replace the three question marks with two variables of data type INT.
3. Drag the element **Branch** from the **Toolbox** view into your implementation and release the mouse button at the green insert position directly at the output of the operator.
 - ⇒ The line branch splits the processing line at the output of the operator box into two subnetworks. Further FBD elements or line branches can be added to each of the two subnetworks.

See also:

- TC3 User Interface documentation: [Command Insert Network \[► 1025\]](#)

7.5.1.2 Programming Ladder Diagrams (LD)

Creating a POU in the Ladder Diagram implementation language (LD)

1. Select a folder in the **Solution Explorer** in the PLC project tree.
2. In the context menu select the command **Add > POU...**
 - ⇒ The **Add POU** dialog opens.
3. Enter a name and select the implementation language Ladder Diagram (LD).
4. Click on **Open**.

⇒ TwinCAT adds the POU to the PLC project tree and opens it in the editor. An empty network is added in the implementation part. On the left, the empty network is limited by a vertical line, which represents a busbar. The **Toolbox** view opens automatically, providing suitable elements, operators and function blocks for LD programming.

Adding a contact and a function block (TON)

- ✓ A POU with the implementation language LD is opened in the editor, and an empty network is added.
 1. Click on the category **Ladder elements** in the **Toolbox** view.
 2. Click the **Contact** element, drag it into your network and release the mouse button at the insert position **Start here**.
 - ⇒ The contact is added on the left of the network directly at the vertical line.
 3. Click on **???** and enter the name of a Boolean variable. The input assistant is available for this purpose.
 4. In the **Toolbox** view click on the category function blocks and drag the function block TON to an insert position on the connecting line to the right of the added contact.
 - ⇒ TwinCAT adds the function block TON to the right of the contact. The contact is linked to the input IN of the TON function block.
 5. Enter a time constant, e.g. T#3s, at the input PT.
- ⇒ If the variable of your contact becomes TRUE, the input IN of the TON function block also becomes TRUE. The TON function block passes the value TRU to output Q with a switch-on delay of T#3s, for example.

Inserting a closed branch

- ✓ A POU with the implementation language LD is opened in the editor, and an empty network is added.
 1. Click in the empty network and select the command **Insert Contact** in the menu **FBD/LD/IL**.
 2. Select the connecting line to the left of the contact and select the command **Set Branch Start Point** in the menu **FBD/LD/IL**.
 - ⇒ The start point on the connecting line is indicated by a red rectangle. TwinCAT indicates all possible end points of the branch with a blue rectangle.
 3. Click on a blue rectangle to set the end point of your closed line branch.
 - ⇒ TwinCAT adds the line branch between the start and end points. The program will run through both branches up to the end point.
 - If you insert the line branch at a function block instead of a contact, the function block is only called if none the other branches is TRUE.

See also:

- TC3 User Interface documentation: [Command Insert Contact \[► 1030\]](#)
- TC3 User Interface documentation: [Command Set Branch Start Point \[► 1034\]](#)

7.5.1.3 Programming Instruction Lists (IL)



If required, IL can be enabled via the TwinCAT options. (Tools > Options > TwinCAT > PLC programming environment > FBD, LD and IL > IL)

Creating a POU in the Instruction List implementation language (IL)

1. Select a folder in the **Solution Explorer** in the PLC project tree.
2. In the context menu select the command **Add > POU...**
 - ⇒ The **Add POU** dialog opens.
3. Enter a name and select the implementation language Instruction List (IL).
4. Click on **Open**.
 - ⇒ TwinCAT adds the POU to the PLC project tree and opens it in the editor. A network has already been added in the implementation part.

Programming a network (e.g. for an ADD operation)

- ✓ A POU (IL) is opened in the editor with an empty network.
- 1. Click in the 1st column of the highlighted row and enter the operator LD.
- 2. Press the **[Tab]** key.
 - ⇒ The cursor jumps to the 2nd column.
- 3. Enter the first summand of your ADD operation, e.g. “6”.
- 4. Press **[Ctrl] + [Enter]** or select the command **Insert IL line below** in the menu **FBD/LD/IL**.
 - ⇒ TwinCAT adds a new statement line at the bottom. The focus is in the first column of this row.
- 5. Enter ADD and press **[Tab]**.
- 6. Enter the 2nd summand of your ADD operation, e.g. “12”.
- 7. Press **[Ctrl] + [Enter]**
- 8. Enter the operator ST and press **[Tab]**.
- 9. Enter a variable of data type INT, e.g. “nVar”.
 - ⇒ The result, in the example “16”, is stored in nVar.

See also:

- TC3 User Interface documentation: [Command Insert IL line below \[▶ 1032\]](#)

Calling a function block

- ✓ A POU (IL) is opened in the editor with an empty network. A variable with data type <function block>, is declared in the declaration part, e.g. `fbSample : CTU; .`
- 1. Click in the first column of the highlighted row and select the command **Insert Box** in the menu **FBD/LD/IL**.
 - ⇒ The **Input Assistant** opens.
- 2. In the category **Function blocks** or **Module calls** select the required function block, for example the counter CTU from the library Tc2_Standard, and click **OK**.
- 3. TwinCAT adds the selected function block CTU as follows:

```

CAL      ??? (
CU:=    ???,
RESET:= ???,
PV:=    ???,
Q=>     ???,
CV=>    ???)
  
```

- 4. Replace the ??? strings with the variable names and the values or variables for the inputs/outputs of the function block.
- 5. As an alternative to adding the function block via the input assistant, you can enter the call directly in the editor.

See also:

- TC3 User Interface documentation: [Command Insert Box \[▶ 1026\]](#)

7.5.2 Continuous Function Chart (CFC)

The Continuous Function Chart (CFC) implementation language is a graphical programming language and extends the standard IEC 61131-3 languages.

You can insert and freely position elements in a programming block in CFC. With connections you interconnect the elements to a network. You can also insert feedback. The result is a clear function chart that you can read like a circuit diagram or a block diagram.

The execution order of a function chart is data flow based. A programming block can process several data flows. The data flows then have no common data and there are several networks in the editor that have no connections between them. Programming blocks in FBD, LD or IL have a network-based execution order.

The implementation language Continuous Function Chart (CFC) - page-oriented is also a graphical programming language and extends the standard languages of IEC 61131-3.

You can graphically program space-consuming, complex function charts in this language. The same elements and commands are available for this as in Continuous Function Chart (CFC). In addition, you can arrange the code distributed on any number of pages. This allows you to create extensive function charts that are still easy to print. In the border areas of each page, you can arrange inputs and sink connection marks on the left, and outputs and source connection marks on the right. This supports you when inserting connecting lines and helps to get a better overview.

It is not possible to switch a programming block between the implementation language Continuous Function Chart (CFC) - page-oriented and Continuous Function Chart (CFC).

See also:

- Reference Programming: [Continuous Function Chart \(CFC\) and Page-Oriented CFC \[► 666\]](#)
- TC3 User Interface documentation: [CFC \[► 1011\]](#)

7.5.2.1 Programming in CFC

You can wire programming blocks together in the CFC Editor and create descriptive block diagrams.

The editor supports you as follows:

- Programming with elements and connecting lines
- Dragging instances and variables to the editing area
- Auto-routing of the connecting lines
- Automatic connecting
- Fixing connecting lines by control points
- Collision detection
- Input support for connection marks
- Forcing and writing values in online mode
- Moving the selection using arrow keys
- Reduced representation of a function block without unconnected connections

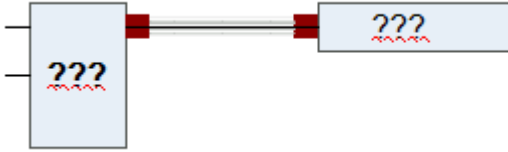
Creating a POU in the implementation language Continuous Function Chart (CFC)

1. Select a folder in the **Solution Explorer** in the PLC project tree.
2. In the context menu select the command **Add > POU...**
 - ⇒ The **Add POU** dialog opens.
3. Enter a name and select the implementation language **Continuous Function Chart (CFC)**.
4. Click **Open**.
 - ⇒ TwinCAT adds the POU to the PLC project tree and opens it in the editor.

Insert elements and interconnect them with connecting lines

1. Position a **Function block** element and an **Output** element in the editor. Use the mouse to drag the elements from the **Toolbox** view into the editor.
2. Click on the output of the **Function block** element.
 - ⇒ The output is marked with a red square.
3. With the mouse button pressed, draw a connecting line from the output of the **Function block** element to the input of the **Output** element.
 - ⇒ When the input pin is reached, the cursor changes its symbol.
4. Release the mouse button.

⇒ The output pin of the function block is connected to the input pin of the output.



Alternatively, you can select the two pins while pressing the **[Ctrl]** key, followed by selecting the command **Connect Selected Pins** in the **CFC** menu or the context menu.

Call instances

1. Create a new project and add a default PLC project
2. Create the function block FB_Sample in the implementation language ST with inputs and outputs:

```
FUNCTION_BLOCK FB_Sample
VAR_INPUT
  nIn1 : INT;
  nIn2 : INT;
  sIn1 : STRING := 'Name';
  bIn1 : BOOL;
END_VAR
VAR_OUTPUT
  nOut : INT;
  sOut : STRING;
  bOut : BOOL;
END_VAR
VAR
  nCounter : INT;
  fbSample1 : FB_Sample;
  fbSample2 : FB_Sample;
  nResult : INT;
  sResult : STRING;
  bResult : BOOL;
END_VAR
```

```
nOut := nIn1 + nIn2;
```

```
sResult := sIn1;
```

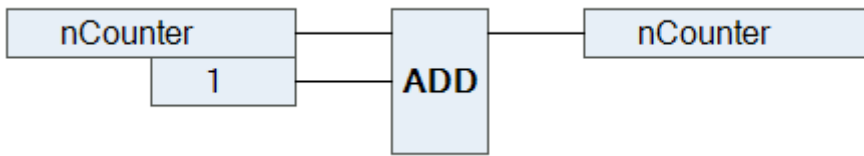
```
IF bIn1 THEN
  bOut := TRUE;
END_IF
```

3. Create the program PRG_First in the implementation language ST .
4. Instantiate function blocks and declare variables:

```
PROGRAM PRG_First
VAR
  nCounter : INT;
  fbSample1 : FB_Sample;
  fbSample2 : FB_Sample;
  nResult : INT;
  sResult : STRING;
  bResult : BOOL;
END_VAR
```

5. Drag a **Box** element from the **Toolbox** view into the editor.
6. Click on the **???** field and type **ADD**.
 - ⇒ The function block type is **ADD**. The function block acts as an adder.
7. Click the line number 3 in the declaration editor.
 - ⇒ The declaration line of **nCounter** is selected.
8. Click in the selection and drag the selected variable into the implementation. Focus an input of the **ADD** function block there.
 - ⇒ An input was created, declared and connected to the function block.
9. Click again in the selection and drag the variable to the output of the **ADD** function block.
 - ⇒ An output was created, declared and connected to the function block.
10. Drag from the **Toolbox** view an **Input** element into the implementation.
11. Click on its field **???** and enter **1**.
12. Connect input 1 to an input on the **ADD** function block.

⇒ A network is programmed. At runtime, the network counts the cycles and stores the result in nCounter.

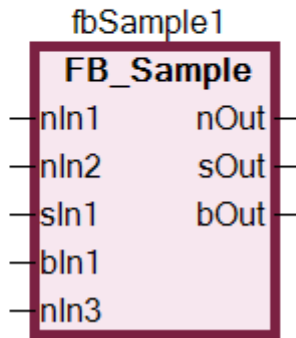
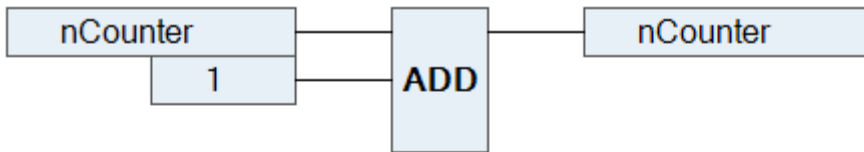


13. Click the line number 4 in the declaration editor.

⇒ The line with fbSample1 is selected.

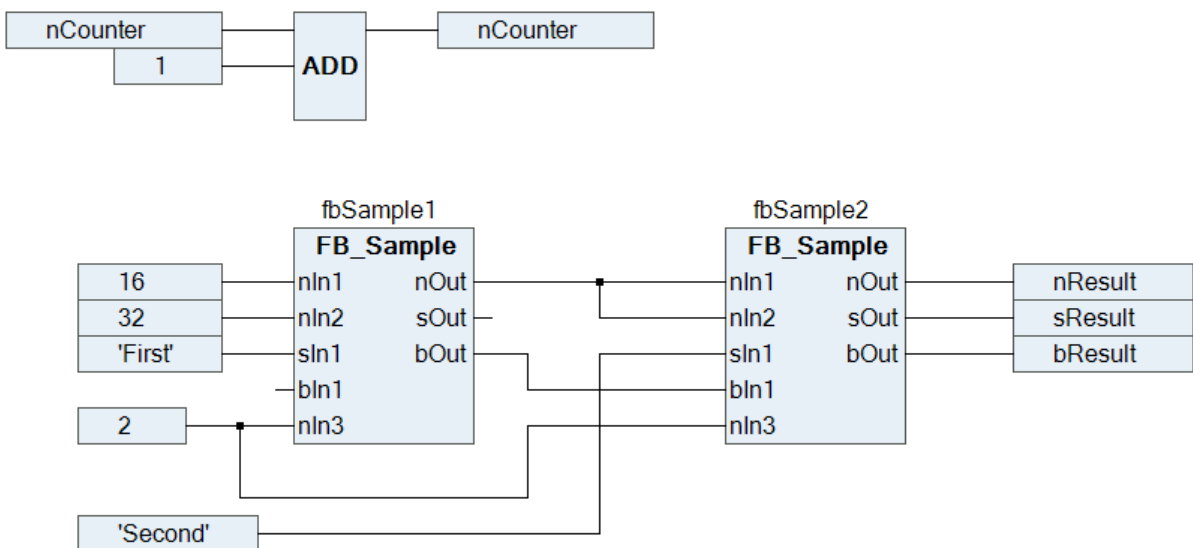
14. Click in the selection and drag the selected instance into the implementation.

⇒ The instance appears as a function block in the editor. Type, name and the function block pins are displayed accordingly.



15. Drag the fbSample2 instance into the editor. Interconnect the instances with each other and with inputs and outputs.

⇒ Sample:



A program in ST with the same functionality could look like this:

```
PROGRAM PRG_First_ST
VAR
  nCounter : INT;
```



```

fbSample1 : FB_Sample;
fbSample2 : FB_Sample;
nResult   : INT;
sResult   : STRING;
bResult   : BOOL;
END_VAR

nCounter := nCounter + 1;
fbSample1(nIn1 := 16, nIn2 := 32, sIn1 := 'First', xItem := TRUE, nIn3 := 2, nOut => fbSample2.nIn1,
bOut => fbSample2.bIn1);
fbSample2(nIn2 := fbSample1.nOut, sIn1 := 'Second', nIn3 := 2, nOut => nResult, sOut => sResult,
bOut => bResult);

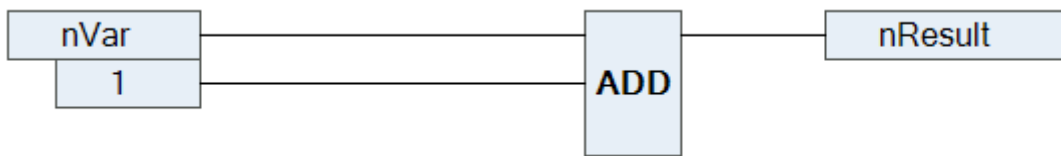
```

Creating connection marks

✓ You have a CFC programming block with connected elements.

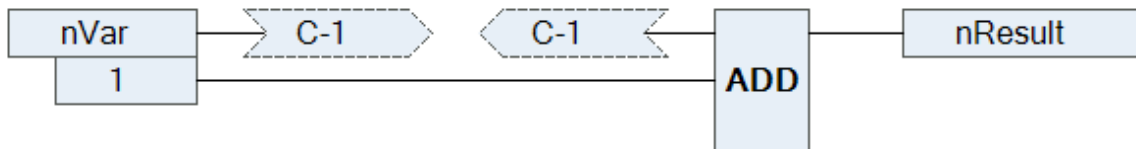
1. Select a connecting line between two elements.

⇒ The connecting line is selected, and the input or output of the elements is marked with a red square



2. Select the command **Connection Mark** in the context menu or the **CFC** menu.

⇒ The connection is disconnected and replaced with a **connection mark - source** and a **connection mark - sink**. The mark name is generated automatically.

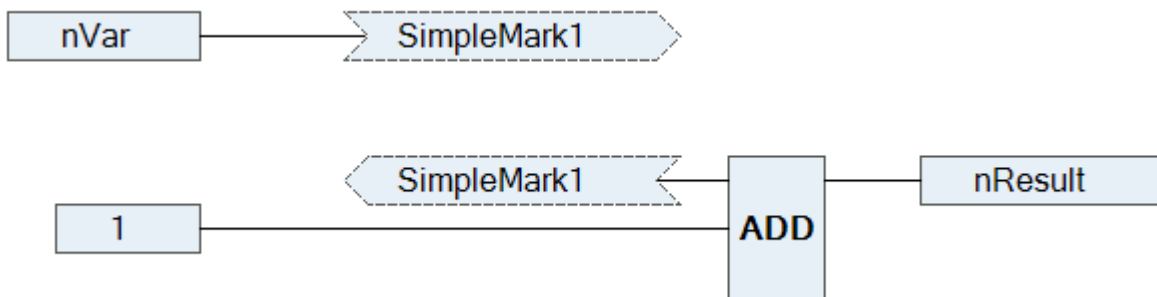


3. Click in the connection mark source.

⇒ The name can be edited.

4. Enter a name for the source connection mark.

⇒ Source connection mark and sink connection mark have the same name.



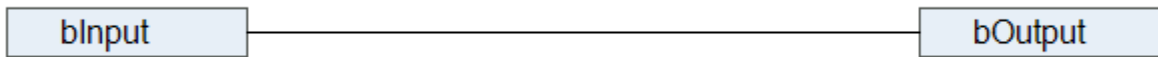
See also:

- TC3 User Interface documentation: [Command Connection Mark \[► 1022\]](#)

Resolving collisions and fixing connecting lines by control points

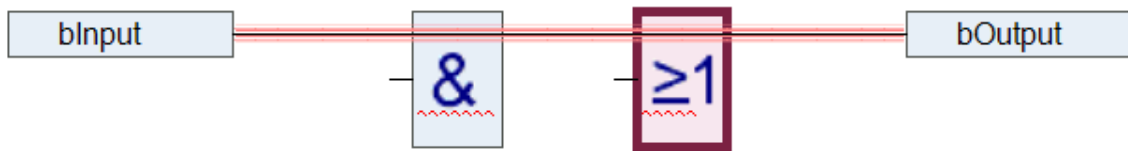
The following example illustrates the application if the command **Route All Connections**, and the application of control points.

1. Position the elements **input** and **output** and link the elements.



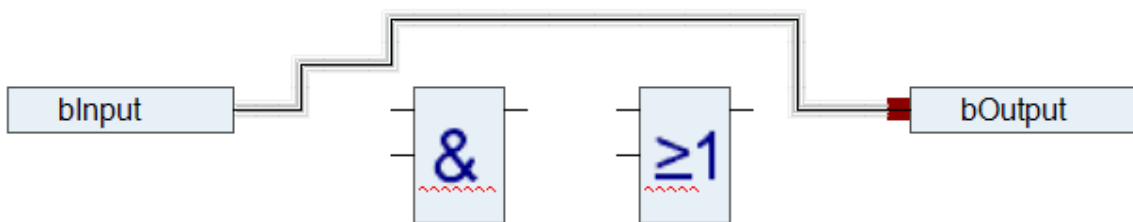
2. Position two **function block** elements on the line.

⇒ The connecting line and the function blocks are shown in red due to the collision.

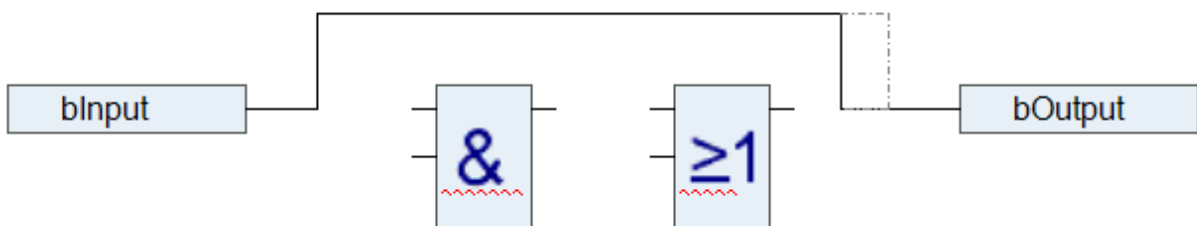


3. Select the command **Route All Connections** in the **CFC > Routing** menu.

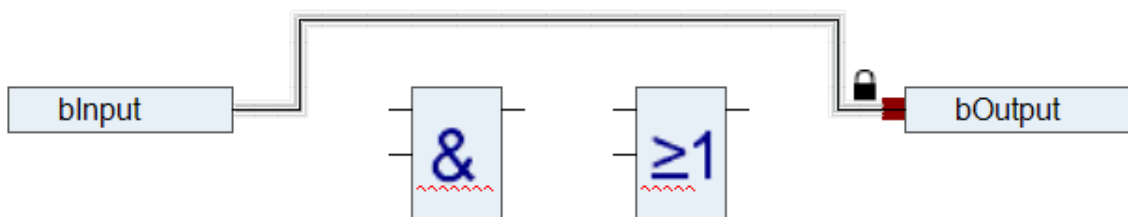
⇒ The collision is resolved.



4. Change the connecting lines step by step.

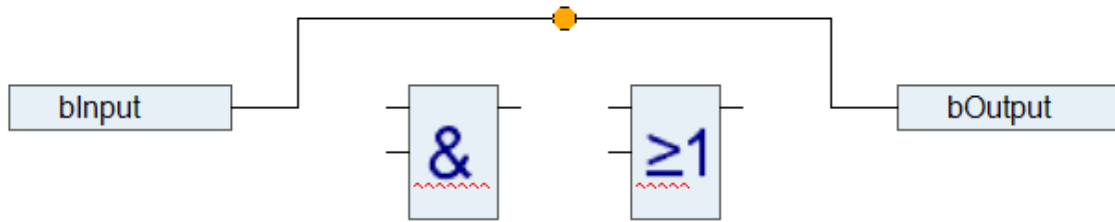


⇒ The connecting line was changed manually and is now locked for auto-routing. This is indicated by a padlock at the end of the connection.

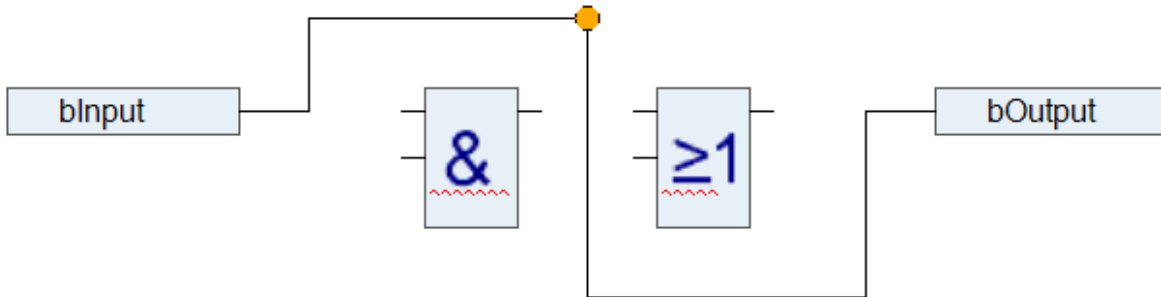


5. Select the connecting line and select the command **Create Control Point** in the **CFC > Routing** menu. Alternatively, you can drag a control point from the **Toolbox** view onto a line.

⇒ A control point is created on the connecting line. The connecting line is fixed at the control point.



6. Change the connection according to the example below.



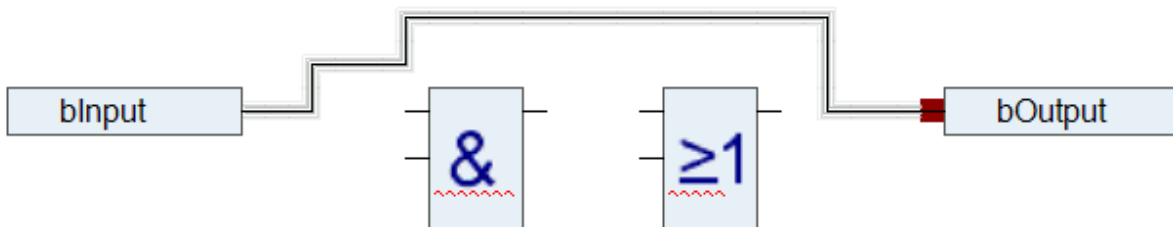
⇒ The control point enables you to modify the connecting line as required. You can set as many control points as you want.

7. Remove the control point with the command **Remove Control Point** in the **CFC > Routing** menu.

8. Unlock the connection with the command **Unlock Connection** or by clicking on the padlock symbol.

9. Select the connecting line and select the command **Route All Connections**.

⇒ The connecting line is drawn automatically, as shown under step 3.



Connections within a group are not automatically routed.

See also:

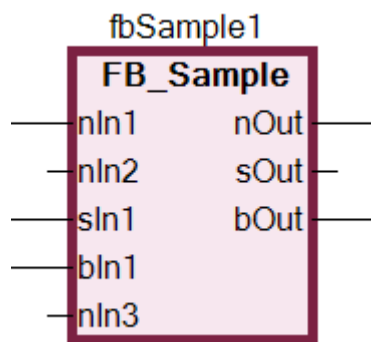
- TC3 User Interface documentation: [Command Route All Connections \[► 1021\]](#)
- TC3 User Interface documentation: [Command Remove Control Point \[► 1021\]](#)
- TC3 User Interface documentation: [Command Unlock Connection \[► 1018\]](#)

Reduce representation of a function block

Requirement: a CFC programming block is open. In the editor, its function blocks are displayed with all declared connections.

1. Select a function block whose connections are partially unconnected.

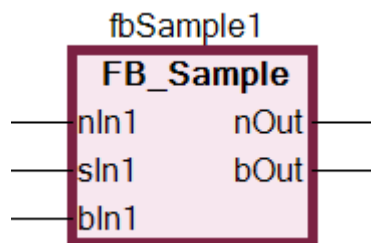
⇒ Example: FB_Sample



⇒ The function block requires space for all connections.

2. From the menu **CFC > Routing**, select **Remove Unused Pins**.

⇒ The function block requires less space and is now only displayed with the functionally relevant connections.



Also see about this

📖 Command Forward by one [▶ 1016]

7.5.2.2 Automatic execution order by data flow

The execution order in programming blocks is uniquely determined in text-based and network-based editors. In the CFC editor, however, you can position the elements freely, so the execution order is initially not unique. It is therefore determined by TwinCAT according to the data flow and, in the case of multiple networks, according to the topological position of the elements. The elements are sorted from top to bottom and left to right. The programming block is thus processed while optimized by time and by cycle.

You can get information about the time sequence of the elements in the chart and briefly display the execution order. When you program networks with feedback, you can define an element as the starting point in the feedback loop.

You can also explicitly edit the processing order in a CFC object if necessary. To do this, change the value of the **Explicit Execution Order** property from the default False to True. In the Explicit Execution Order mode you have the possibility to edit the execution order with menu commands.

Data flow

The chronological sequence of which data is read or written when and how in which programming objects is generally referred to as data flow. A programming block can process any number of data flows, which can also be executed independently of each other.

Displaying execution order

In the automatic data flow mode, which is activated by default, the execution order of a CFC object is determined automatically. You can have the execution order briefly displayed in the CFC editor.

1. Create a new project and add a default PLC project.

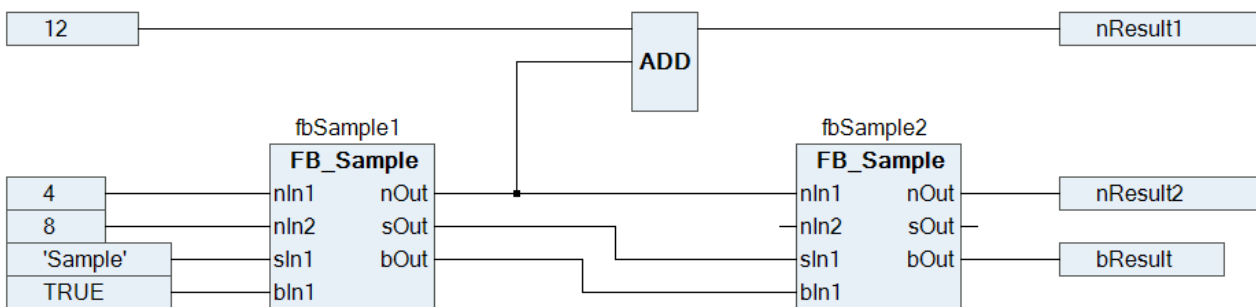
2. Create the function block FB_Sample in the ST implementation language with inputs and outputs.

```
FUNCTION_BLOCK FB_Sample
VAR_INPUT
  nIn1 : INT;
  nIn2 : INT;
  sIn1 : STRING := 'Name';
  bIn1 : BOOL;
END_VAR
VAR_OUTPUT
  nOut : INT;
  sOut : STRING;
  bOut : BOOL;
END_VAR
VAR
  nOut := nIn1 + nIn2;
  sResult := sIn1;
  IF bIn1 THEN
  bOut := TRUE;
  END_IF
END_VAR
```

```
nOut := nIn1 + nIn2;
sResult := sIn1;
IF bIn1 THEN
bOut := TRUE;
END_IF
```

3. Create the Execute_CFC program in the CFC implementation language.

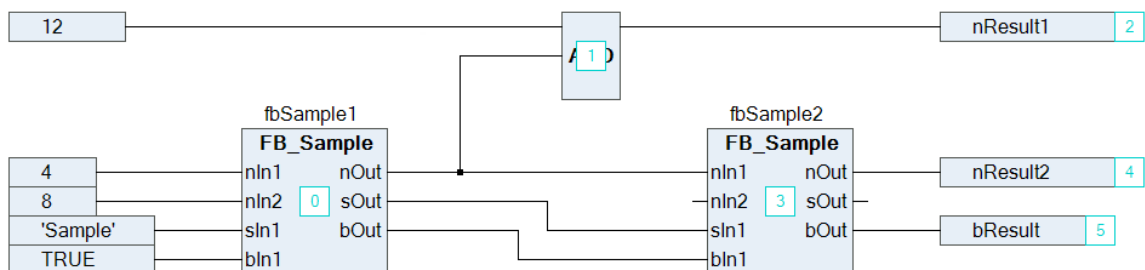
```
PROGRAM Execute_CFC
VAR
  fbSample1 : FB_Sample;
  fbSample2 : FB_Sample;
  nResult1 : INT;
  nResult2 : INT;
  bResult : BOOL;
END_VAR
```



Newly created programming objects in CFC have the automatic data flow mode activated. The execution order of the programming object is optimally defined internally.

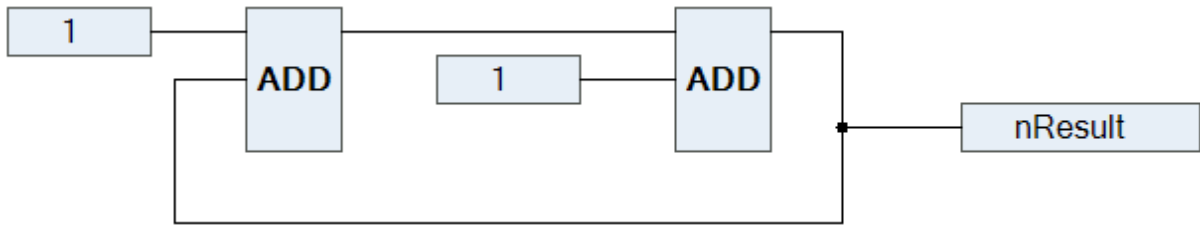
4. Select the command **Execution Order > Display Execution Order** in the **CFC** menu.

⇒ The execution order of the object is shown. The boxes and outputs are numbered accordingly and reflect the chronological processing order. As soon as you click again in the CFC editor, the numbering will be hidden.



Determining execution order in feedback networks

1. Create a CFC program with feedback.




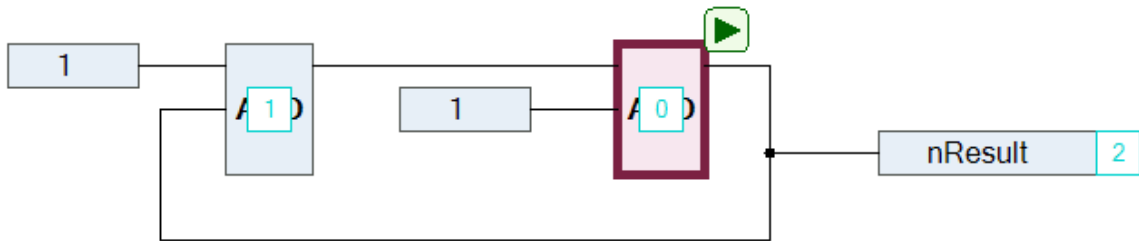
2. Select an element within the feedback.

⇒ The selected element is highlighted in red.

3. Select the command **Execution Order > Set Start of Feedback** in the menu **CFC**.

⇒ At runtime, this function block is processed first. The start function block of the feedback is defined

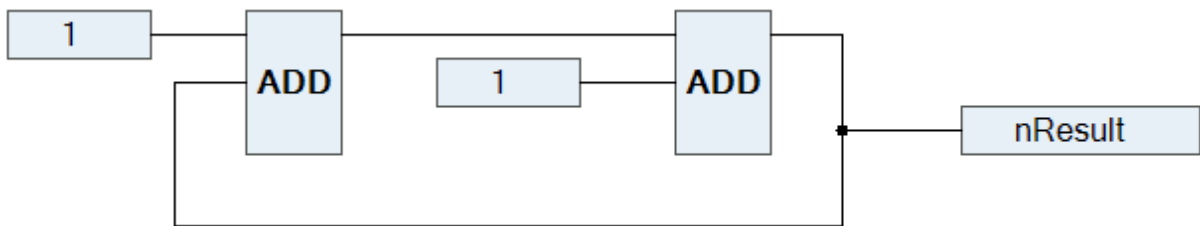
and decorated with the  symbol. The execution order is reordered and the selected element gets the number 0 (generally the lowest number in the feedback).



4. Select the start function block again.

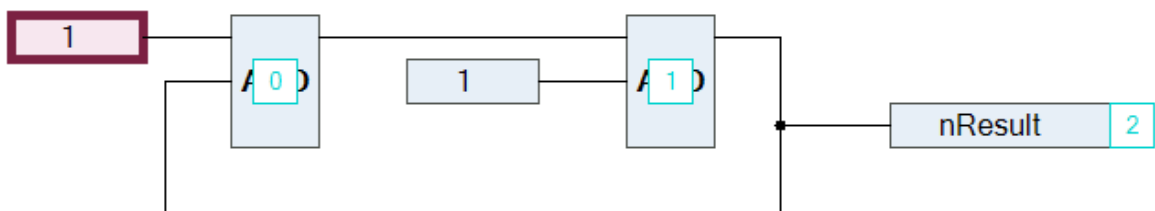
5. Select the command **Execution Order > Set Start of Feedback** in the menu **CFC**.

6. The function block is deselected as start function block. The execution order is defined internally.



7. Select the command **Execution Order > Display Execution Order** in the **CFC** menu.

⇒ The execution order by data flow is displayed.



Defining the execution order explicitly

The automatically defined execution order by data flow results in time and cycle optimized execution of the programming block. Thus, you do not need any information about the internally managed execution order during the development process.

In the Explicit Execution Order mode, you adjust the execution order on your own responsibility and have to assess the consequences and effects yourself. This is another reason why the execution order is always displayed.

You can explicitly change the automatically defined execution order of a CFC object if you have enabled the Explicit Execution Order property.

1. Select a CFC object in the project tree.
2. In the context menu, click **Properties**.
3. Set the **Explicit Execution Order** option under CFC to True.
 - ⇒ The Explicit Execution Order mode is enabled. In the CFC editor, the networks are numbered. Various commands for editing the execution order are available at **Execution order** in the **CFC** menu.
4. Open the CFC object.
5. Select a numbered element and select the command **Execution Order > Move to Beginning**.
 - ⇒ The execution order is resorted and the selected element has the number 0.

See also:

- TC3 User Interface documentation: [Command Forward by one](#) [▶ 1016]

7.5.3 Structured Text (ST), Extended Structured Text (ExST)

The ST Editor is used for programming POU's in the IEC-61131-3 programming language "Structured Text (ST)" or "Extended Structured Text". "Extended Structured Text" offers certain additional functions, compared with the standard IEC 61131-3 language.

Structured Text is a programming language that is comparable to other high-level languages such as C or PASCAL, which enables the development of complex algorithms. The program code consists of a combination of expressions and instructions, which may be conditional (IF...THEN...ELSE) or looped (WHILE...DO).

An expression is a construct that returns a value after it was evaluated. Expressions can also be a combination of operators and operands. Assignments can also be used as expressions. An operand can be a constant, a variable, a function call or another expression.

Instructions control how the expressions are to be processed.

For this text editor various settings are available in the **Options** and **Customize** dialogs of the **Tools** menu with regard to behavior, appearance and menus. The familiar Windows functions (e.g. IntelliMouse) are also available for this editor.

ExST - Extended Structured Text

"Extended Structured Text (ExST)" is a TwinCAT-specific extension for "Structured Text (ST)", as defined in the IEC 61131-3 standard.

See also:

- TC3 User Interface documentation: [Command Customize](#) [▶ 995]
- TC3 User Interface documentation: [Command Options](#) [▶ 966]
- Reference Programming: [Structured Text and Extended Structured Text \(ExST\)](#) [▶ 624]

7.5.3.1 Programming Structured Text (ST)

Principle

The ST Editor can be used for programming in the programming languages Structured Text and Extended Structured Text. The program code consists of a combination of expressions and instructions, which may be conditional or looped. Each instruction has to finish with a ;

Variables are declared in the declaration editor.

See also:

- [Using the Declaration Editor \[► 70\]](#)

Creating a POU in the implementation language Structured Text (ST)

1. Select a folder in the **Solution Explorer** in the PLC project.
2. In the context menu select the command **Add > POU...**
 - ⇒ The **Add POU** dialog opens.
3. Enter a name and select the implementation language "Structured Text (ST)".
4. Click on **Open**.
 - ⇒ The POU is added in the PLC project tree and opened in the editor. Now add the variable declarations in the upper part of the POU and the ST program code in the lower part of the POU.

7.5.4 Sequential Function Chart (SFC)

Use the SFC editor to program POU's in the IEC 61131-3 programming language Sequential Function Chart. Sequential Function Chart (SFC) is a graphical language, which facilitates description of the chronological sequence of individual actions in a program. To this end the actions, which are separate programming objects, are assigned to the step elements. The step processing sequence is controlled by transition elements.

See also:

- Reference Programming: [Sequential Function Chart \(SFC\) \[► 635\]](#)
- TC3 User Interface documentation: [SFC \[► 1001\]](#)

7.5.4.1 Programming in Sequential Function Chart (SFC)

Creating programming block in the SFC implementation language

1. Select a folder in the **Solution Explorer** in the PLC project.
2. In the context menu select the command **Add > POU...**
 - ⇒ The **Add POU** dialog opens.
3. Enter a name and select the implementation language Sequential Function Chart (SFC).
4. Click **Open**.
 - ⇒ TwinCAT adds the programming block to the PLC project tree and opens it in the editor.

Adding step-transitions

1. Select the transition after the Init step.
 - ⇒ The transition is shown in red.
2. In the **SFC** menu or the context menu select the command **Insert step-transition after**.
 - ⇒ TwinCAT adds step Step0 and transition Trans0.
3. Select the transition Trans0 and in the **SFC** menu or the context menu select the command **Insert step transition**.
 - ⇒ TwinCAT adds transition Trans1 and step Step1 before Trans0.

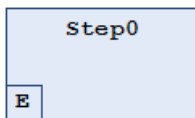
Alternatively, you can move the elements **Step** and **Transition** from the **Toolbox** view into the diagram with drag and drop.

See also:

- TC3 User Interface documentation: [Command Insert step-transition after \[► 1002\]](#)
- TC3 User Interface documentation: [Command Insert step transition \[► 1001\]](#)

Add entry action

1. Select Step0.
2. In the **SFC** menu or the context menu select the command **Add entry action**.
 - ⇒ By default, a prompt appears for specifying the duplication mode for the step actions. This is used to specify whether the reference information is copied to the existing step action objects or whether the objects should be "embedded" when the step is copied in the future. If embedding is selected, new step action objects are created when the step is copied. The duplication mode is defined in the step property **Duplicate or Copy**. As long as this property is disabled, the copied steps call the same actions as the current step.
3. For this example, retain the **Copy reference** default setting and confirm with **OK**.
 - ⇒ The **Add entry action** dialog opens.
4. Enter "Step0_entry" as name and select the implementation language "Structured Text (ST)". Click **Add**.
 - ⇒ TwinCAT adds the action Step0_entry under the function block in the PLC project tree and opens the action in the editor. In the entry action Step0_entry you can now program instructions, which are to be executed once when the step Step0 is enabled.
5. Close the editor for Step0_entry.
 - ⇒ The step Step0 is now marked with an E in the lower left corner. Double-clicking the E to open the editor.



- ⇒ The entry action Step0_entry now appears in the step properties under **Entry action**. Here you can also select another action, if required.
6. Select Step0. Press **[Ctrl] + [C]** to copy the step.
 - ⇒ The added copy of the step contains the same entry action that was added above. In other words, the new step calls the same action.

● **SFC editor options**

I In the TwinCAT options for the **SFC editor** category you can specify whether the prompt for specifying the duplication mode should always appear when a step action is added, or you can specify duplication mode as standard.

See also:

- Reference Programming: [SFC Element Properties \[► 650\]](#)
- TC3 User Interface documentation: [Command Add entry action \[► 1007\]](#)
- TC3 User Interface documentation: [Dialog Options - SFC editor \[► 977\]](#)

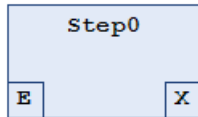
Add exit action

1. Select Step0.
2. In the **SFC** menu or the context menu select the command **Add exit action**.
 - ⇒ By default, a prompt appears for specifying the duplication mode for the step actions. Please refer to the notes in sections "Adding an input option" and "SFC element properties".
 - ⇒ The **Add exit action** dialog opens.
3. Enter "Step0_exit" as name and select the implementation language "Structured Text (ST)". Click **Add**.

⇒ TwinCAT adds the action Step0_exit under the function block in the PLC project tree and opens the action in the editor. In the exit action Step0_exit you can now program instructions, which are to be executed once before the step Step0 is disabled.

4. Close the editor for Step0_exit.

⇒ The step Step0 is now marked with an X in the lower right corner. Double-clicking the E to open the editor.



⇒ You can define the exit action in the step properties under exit action. Here you can also select another action, if required.

See also:

- TC3 User Interface documentation: [Command Add exit action \[► 1008\]](#)

Add Action

1. Double-click on Step0.

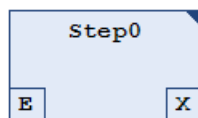
⇒ By default, a prompt appears for specifying the duplication mode for the step actions. Please refer to the notes in sections “Adding an input option” and “SFC element properties”. The **Add Action** dialog opens.

2. Enter "Step0_active" as name and select the implementation language "Structured Text (ST)". Click **Add**.

⇒ TwinCAT adds the action Step0_active under the function block in the PLC project tree and opens the action in the editor. In the step action Step0_active you can now program instructions, which are to be executed as long as the step is active.

3. Close the editor for Step0_active.

⇒ Step0 is now marked with a black triangle in the upper right corner.



You can define the action in the properties of the step under Step actions. Here you can also select another action.

Adding an alternative branch

1. Select Step1.

2. In the **SFC** menu or the context menu select the command **Insert branch right**.

⇒ TwinCAT adds the step Step2 to the right of Step1. The steps are as linked as parallel branch with a double line.

3. Select one of the two double lines.

⇒ The double line is shown in red.

4. Select the command **Alternativ** in the context menu or the **SFC** menu.

⇒ TwinCAT converts the branch to an alternative branch. The double line changes to a single line.

An alternative branch can be converted to a parallel branch with the **Parallel** command.

See also:

- TC3 User Interface documentation: [Command Insert branch right \[► 1003\]](#)
- TC3 User Interface documentation: [Command Alternative \[► 1002\]](#)


Adding a jump

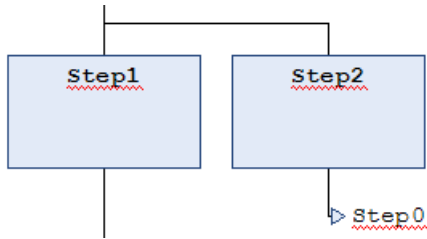
1. Select Step2.

2. In the **SFC** menu select the command **Insert jump after**.

⇒ TwinCAT adds the Step jump after step Step2.

3. Click on the jump destination **Step** of the jump.

⇒ You can now enter the jump destination manually or select it via the Input Assistant . Select Step0.



See also:

- TC3 User Interface documentation: [Command Insert jump after \[► 1006\]](#)

Adding macro

1. Select Step1.

2. In the **SFC** menu or the context menu select the command **Insert macro after**.

⇒ TwinCAT adds the macro Macro0 after the step Step1.

3. Double-click on the element Macro0.

⇒ The macro opens in the implementation part of the editor. The header shows the name Macro0.

4. In the **SFC** menu or the context menu select the command **Insert step transition**.

⇒ TwinCAT inserts a step transition combination.

5. In the **SFC** menu or the context menu select the command **Exit macro**.

⇒ The implementation part shows the main diagram again.

See also:

- TC3 User Interface documentation: [Command Insert macro after \[► 1006\]](#)
- TC3 User Interface documentation: [Command Insert step transition \[► 1001\]](#)
- TC3 User Interface documentation: [Command Exit macro \[► 1007\]](#)

Adding an association

1. Select Step2.


2. In the **SFC** menu or the context menu select the command **Insert action association**.

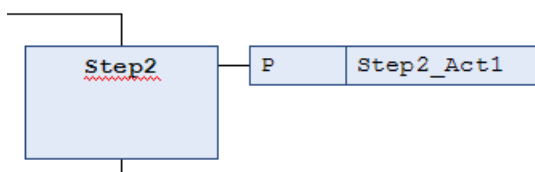
⇒ TwinCAT adds an association to right next to the step Step2.

3. Click in the left field of the association to select the qualifier.

⇒ You can enter the qualifier manually or select it via the Input Assistant . Select "P".

4. Click in the right field of the association to select the action.

⇒ You can enter the action manually or select it via the Input Assistant .



See also:

- TC3 User Interface documentation: [Command Insert action association \[► 1004\]](#)

Analyzing expressions with AnalyzeExpression

The AnalyzeExpression function block from the Tc2_System library allows the analysis of expressions. It can be used in the SFC diagram, for example, to examine the result of the SFCErrror flag, which is used to monitor timeouts in the flow chart.

See also:

- Documentation Tc2_System: AnalyzeExpression
- Reference Programming: [SFC Flags \[► 639\]](#)

7.5.5 Ladder Diagram (LD) (Beta)



The new Ladder editor is available in a beta version from TC3.1 Build 4026.

The Ladder editor is a network-based editor for the IEC programming language Ladder Diagram (LD). It is the successor of the still available FBD/LD editor. The main difference to the FBD/LD editor is the simplified editing possibilities. Variants of elements which previously required separate elements to be inserted from the toolbox are now created simply by the selected insertion position or by subsequently switching a modifier. Programming blocks created in the FBD/LD editor can be converted to Ladder format via a menu command.

See also:

- Chapter Reference Programming: [Ladder Diagram \(LD\) \(Beta\) \[► 680\]](#)
- User interface documentation: [Ladder editor \[► 1038\]](#)

7.5.5.1 Programming in the Ladder editor

The prerequisite for programming in Ladder:

You have created a programming block in the Ladder implementation language in your project. You have opened this box for editing in the Ladder editor.

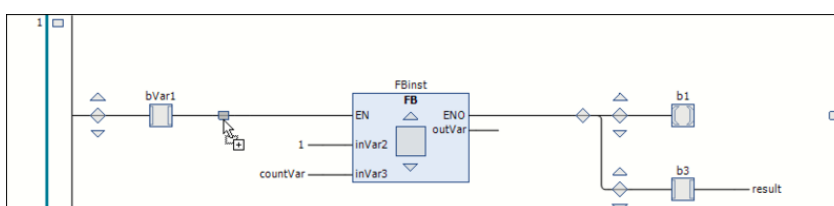
The following instructions describe only individual actions when working in the editor. No specific program is created. If needed, also see the following chapter: [Navigation in the Ladder editor \[► 682\]](#)

Inserting, repositioning, and modifying elements in a network

Insert elements into networks:


1. For example, in the toolbox (Tools window) of the editor, select the **Contact** element.
2. Drag it to the network with the mouse and hold the mouse button.
 - ⇒ When you drag across the network, possible insertion positions are indicated by the following symbols:
 Square with gray background inside an existing element symbol
 Rhombus on a connecting line
 Triangle, pointing down or up, for insertion above or below

3. You can insert the contact element when the mouse cursor gets a plus symbol .



4. Release the mouse button at such a position.
 - ⇒ The contact is inserted in the processing line of the network.

Insert boxes:

1. Add an element **Box** in the network.
 2. Replace the three question marks ??? in the element box with the name of the desired box from the project or a library.
 3. To do this, double-click the question marks and ideally use the Input Assistant by clicking the  button.
- ⇒ The box is inserted and given the appropriate name.

Update boxes:

For example, if you call a function block from your project in the Ladder, you must also update in the ladder diagram after a change to the base FB.



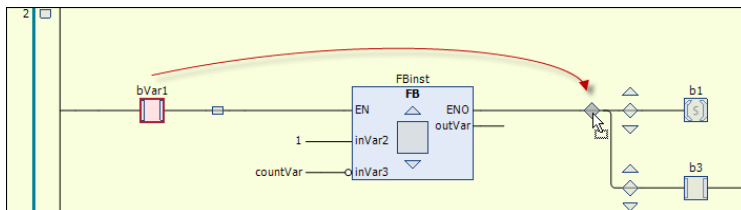
A corresponding "Update" command is currently not yet available.

Insert inputs:

1. Insert an **input** before the box.
 2. To assign variables, replace the three question marks ??? at the element with the variable name.
- ⇒ If necessary, the standard dialog **Auto Declare, Select, Reposition** opens.

Reposition elements in the network:

1. Select an element so that it is displayed with a red background. In the case of a box, the square inside the box icon has to be displayed with a red background.
2. Drag the element to another possible insertion position until the cursor symbol gets a square and release the mouse button.



⇒ The element is positioned accordingly.

Replace elements:

1. To replace an existing element with another, drag the new element onto the one you want to replace.
 2. Release the mouse button when the element to be replaced itself lights up green as an insertion point.
- ⇒ The element is replaced.

Add new network:

1. Drag the Network element to the implementation part.
- ⇒ You will get rectangular insertion marks in the left margin area where the network numberings are displayed. The new network is inserted above the selected insertion point.

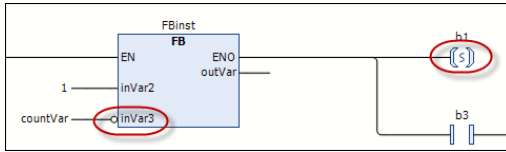


Modify networks:

Provide elements in the network with modifiers:

1. Select the input pin of a box.
2. Right-click to open the context menu.
3. Select the Negate [[▶ 1038](#)] command to negate an input, or one of the Set/Reset [[▶ 1039](#)] or Edge Detect commands (Command: Edge Detection: Rising Edge [[▶ 1039](#)], Command: Edge detection: Falling edge [[▶ 1039](#)]).

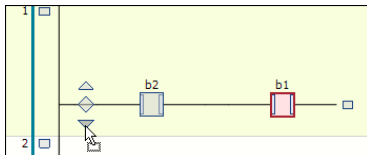
- ⇒ The input is assigned the corresponding symbol.
Example: Negated input on the box, set coil



Creating and editing line branches

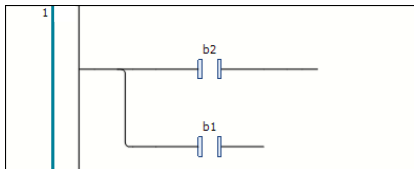
Unlike the LD/FBD editor, there are no elements for line branches in the Ladder editor. You create and edit branches simply by positioning or repositioning elements. Examples:

- 1. Create the following network:



- 2. Drag the second contact to the insertion position marked with a downward pointing arrow in front of the first contact.

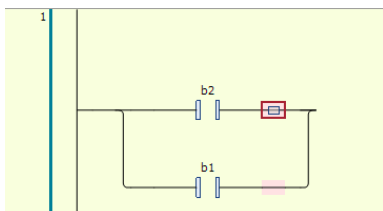
- ⇒ A parallel branching open to the rear is formed. Each branch contains one contact.



To create a closed line branch from the open line branch, i.e. to program an OR construct, proceed as follows:

1. Select both branches behind the contact element (multiple selection).
2. The selection is indicated by the small square with a red background on the line.
3. Then select the Close parallel branch [[▶_1039](#)] command.

- ⇒ The two parallel branches open at the end become a closed branch.



There are two ways to reopen the closed branch:

- You select one of the two branches and drag the selection box to that of the other branch.
- You select both branches and select the command Open parallel branch [[▶_1038](#)].

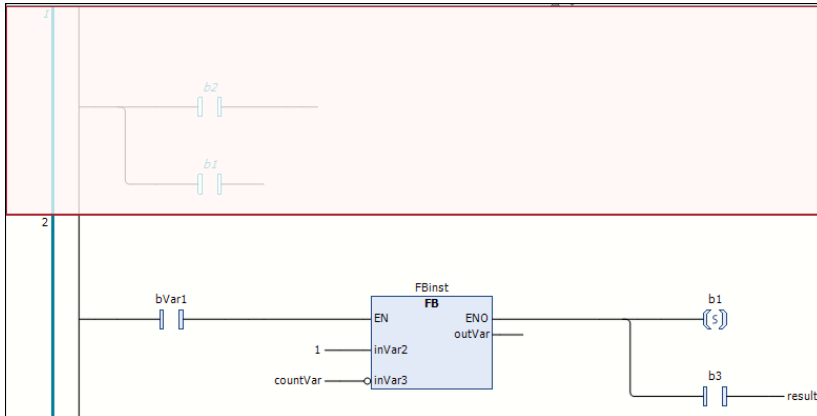


If closed line branches have several parallel branches, the **Open parallel branch** command will always open all branches.

Comment out network

1. Select a network so that it is completely displayed with a red background.
2. Select the Commented out [[▶_1038](#)] command from the context menu.

⇒ The network contents are displayed in gray and the texts in italics. The network is not taken into consideration during processing.

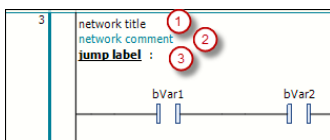


Add network title, network comment, jump label

You can add a network title (1), a network comment (2), and/or a jump label (3) to a network.

1. To do this, click in the first, second, or third line in the upper left corner of the network. A jump label serves as a target when inserting a [jump \[► 1042\]](#) element.

⇒ The display of network title and network comment is defined in the TwinCAT options, category [Ladder \[► 983\]](#).



Also see about this

- Command Outcommented [► 1038]
- Command Insert Jump [► 1042]
- Dialog: Options – Ladder editor [► 983]

7.6 Creating a referenced task

You can integrate one or several tasks in a PLC project to define the processing of program blocks. In order to be able to use a task, you have to create a task reference.

7.6.1 Object Referenced Task

Symbol:

Via the task reference you define the program blocks to be executed in a task.

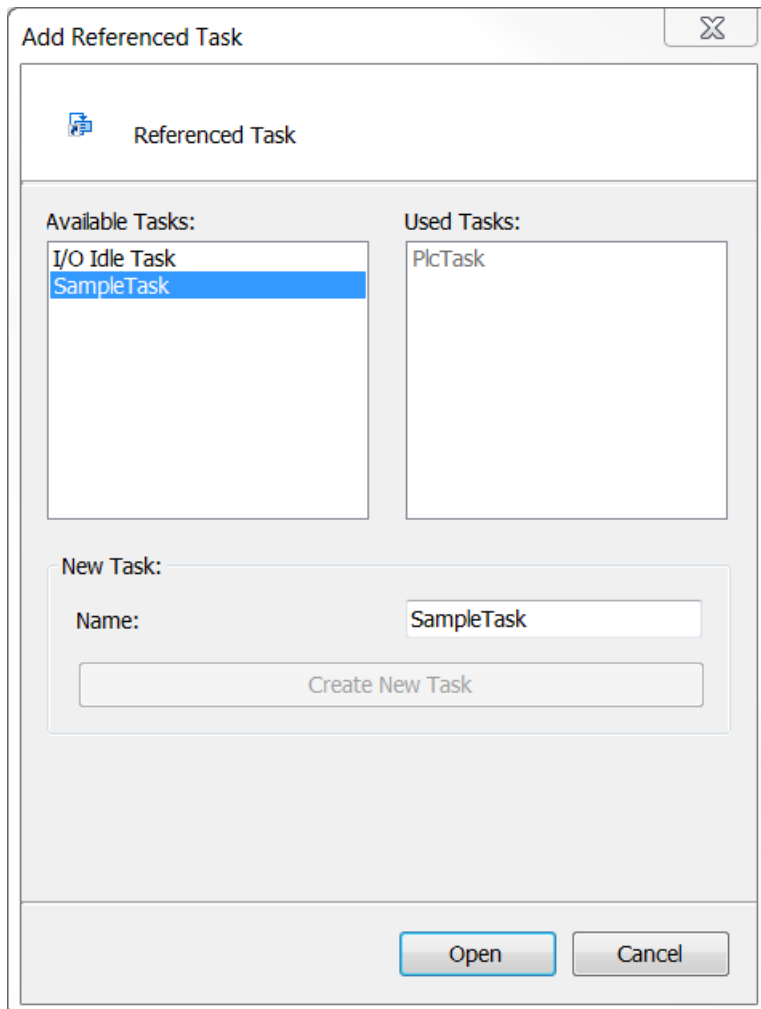
When a standard PLC project is created, a task reference **PlcTask** is created automatically, which defines the processing of the MAIN program block.

Creating an object Referenced Task

1. Select the PLC project (<Project name> Project) in the Solution Explorer in the PLC project tree.
2. In the context menu select the command **Add > Referenced Task**.
 - ⇒ The dialog **Add Referenced Task** opens. The existing tasks are displayed in the **Available Tasks** listbox. The listbox **Used Tasks** shows the already referenced tasks.
3. To generate and integrate a new task, enter a name in the **Name** field and click on **Create New Task**.

- ⇒ The new task is shown in the **Available Tasks** listbox. At the same time the new task appears in the TwinCAT project tree in the **SYSTEM** section under the **Task** node.
- 4. Click on **Open**.
- 5. If the task you wish to integrate already exists, select it in the **Available Tasks** listbox and click **Open**.
- ⇒ The task reference is added to the PLC project tree. When the dialog **Add Referenced Tasks** is opened again, the task is shown in the **Used Tasks** listbox.

Dialog Add Referenced Tasks



Available Tasks	Tasks that are not referenced in the PLC project.
Used Tasks	Tasks that are referenced in the PLC project.

New Task

Name	Name of the new task to be created
Create New Task	Creates a new task

Specifying program blocks in the task

1. Select the task reference in the PLC project tree. In the context menu select the command **Add > Existing Item...**
 - ⇒ The **Input Assistant** dialog opens.
2. Select the program block to be executed in the task and confirm the dialog with **OK**.
3. Alternatively, you can use the mouse to drag the program block in the PLC project tree directly to the task reference.
 - ⇒ The program block appears in the PLC project tree below the task reference.

4. To view the task configuration, double-click on the task reference in the PLC project tree. In the Solution Explorer, the corresponding task is enabled in the TwinCAT project tree in section **SYSTEM > Tasks**. Double-click on the task again to open the task configuration in an editor.
 - ⇒ In the PLC project tree the program blocks that are called by the task are shown under the task reference. The task can be configured in the **SYSTEM** section of the task editor.

Important notes for multitasking systems

The ability of multi-core systems to execute several commands independent of each other can result in a priority reversal.

Example:

A program block that is called by a task with low priority can be assigned to an independent core. In this case, the execution of the task with the lower priority may be completed before a task with higher priority.

7.7 Creating a class diagram

Further information can be found in the documentation "[TF1910 TC3 UML](#)" in section "Creating a new class diagram".

7.8 Configuring the memory reserve for Online Change

You can configure a memory reserve for the online change for function blocks. This means that, provided the memory reserve is sufficiently large, after changes have been made to the declaration of a function block during the subsequent online change, the instances of the function block do not have to be copied to new storage spaces. This refers primarily to online changes in which one or more new variables are added to a function block. If the function block instances do not have to be copied to new storage spaces due to the memory reserve, the online change is faster and fewer problems occur. When the memory reserve is exhausted, a message appears before the online change is performed.

● Configuration of the memory reserve before the first download

i It is best to configure the memory reserve for a function block before downloading the PLC project to the controller for the first time. If the memory reserve is only configured when the PLC project is already on the controller, a complex online change is necessary.

● Alternative configuration option

i As an alternative to the **Online Change Memory Reserve Settings** window, the memory reserve per function block can also be configured in the properties window. To do this, the function block must be selected in the project tree.

Configuring a memory reserve for a function block for the online change

- ✓ In the future, major changes are to be made to a function block of the project. This would require copying the function block instances to other storage locations during online change.
 - ✓ Ideally, the open project is not yet on the controller.
1. From the **PLC** menu, select **Window > Online Change Memory Reserve Settings**.
 - ⇒ The **Online Change Memory Reserve** view opens.
 2. Select the PLC project from the selection list.
 3. From the **Build** menu, choose **Build**.
 4. Click the **Browse application** button.
 5. Select the "All" entry in the "Function blocks" area.
 - ⇒ All function blocks of the PLC project are displayed in the view.
 6. Select the function blocks for which you want to configure a memory reserve.
 - ⇒ If the application is not yet on the controller, the input field "Memory reserve (in bytes)" can be edited.

7. If a PLC project is already on the controller, click on the **Allow** button in the "Allow editing" area.
Note that if you change the memory reserve of a PLC project that is already on the controller, the instances of all affected function blocks must be copied in memory.
8. Enter the size of the memory reserve in bytes and click **Apply to selection**.
⇒ The number of bytes entered is displayed in the table in the **Memory reserve** field.
9. From the **Build** menu, choose **Build**.
10. Click the **Browse application** button.
⇒ In the function block list for the configured function block, the information "Size", "Number of instances", "Additional memory for all instances" and "Remaining memory reserve size" are updated.
11. Load the PLC project into the controller.
⇒ The function block instances occupy the currently required memory and additionally the memory reserve. Future major changes to the function blocks can thus be loaded to the controller via the online change without all instances of the function blocks having to be recopied in the memory.

See also:

- Documentation TC3 User Interface: Reference User Interface > PLC > Window > [Command Online Change Memory Reserve Settings \[► 949\]](#)
- [Performing an Online Change \[► 251\]](#)

7.9 Calling a function block, function or method with external implementation

● Only applicable in exceptional cases

I The use of this functionality is only possible in special constellations. As a rule, you can ignore the external implementation option.

A runtime system may contain an implementation of a function block, function or method, for example from a library. If you create a POU of the same name with the property **External implementation without implementation** in your PLC project, you can execute the existing implementation. Make sure you use an external function block for declaring local variables. An external function or method may not contain local variables.

During the project download, TwinCAT finds the corresponding implementation in the runtime system for each external POU and links it.

Creating a POU with external implementation

1. In the context menu of the PLC project tree select the command **Add > POU...**
2. Enable the function block or function and enter the name of the corresponding runtime system implementation. Quit the dialog with **Open**.
⇒ The POU with the name of the runtime system POU has been created.
3. Select the POU and enable the **Properties** view.
4. Enable the option **External implementation** (late linking in the runtime system).
⇒ The POU is declared, and you can implement a call of the POU.

Creating a method with external implementation

1. Select a function block in the PLC project tree in the **Solution Explorer**.
2. In the context menu select the command **Add > Method** and enter the name of the corresponding runtime system implementation. Quit the dialog with **Open**.
⇒ The method is created.
3. Select the method and enable the **Properties** view.
4. Enable the option **External implementation** (late linking in the runtime system).
⇒ The method is declared, and you can implement a call of the method.

7.10 Using the input wizard

TwinCAT offers functionalities and wizards that facilitate code input during programming.

Dialog Input Assistant

The dialog offers all programming elements, which you can add at the current cursor position. Open the **Input Assistant** dialog with the command **Input Assistant** in the **Edit** menu, from the context menu, or with the keyboard shortcut **[F2]**.

See also:

- TC3 User Interface documentation: [Command Input Assistant \[▶ 873\]](#)

Auto Declare dialog

A dialog provides support for declaring variables. Open the **Auto Declare** dialog with the **Auto Declare** command in the **Edit** menu or via the context menu.

See also:

- TC3 User Interface documentation: [Command Auto Declare \[▶ 875\]](#)

List components

The function **List components** is an input support feature in the text editor, which facilitates entering valid identifiers. The function can be enabled as follows:

1. Select the command **Options** in the **Tools** menu and then the category **TwinCAT > PLC Environment > Smart coding**.
2. Enable the option **List components after typing a dot (.)**.
 - If you then enter a dot instead of global variable, a selection list with all available global variables appears. Double-click on a variable in the selection list or press the **[Enter]** key to add the selected variable after the dot.
 - If you enter a dot instead of a global variable or after a function block-instance variable or a structure variable, TwinCAT shows a corresponding selection list all global variables, all input and output variables of the function block or all structure components. Double-click on a variable in the selection list or press the **[Enter]** key to add the selected variable after the dot. If you also want to see the local variables of function block instances, in the TwinCAT options for **smart coding** enable the option **List all instance variables in the input assistant**.
 - If a component access (with dot) has already been made for a selection list, the last selected entry will be preselected for the next component access.
 - If you enter any string and then press **[Ctrl] + [space bar]**, a selection list appears with all available POU's and global variables. The first element in this list, which starts with the previously entered string, is automatically selected, and you can enter it in the editor by double-clicking or via the **[Enter]** key. Matches with the entered string are highlighted in yellow in the selection list. When the entered string is changed, the displayed selection list is updated.
 - In the ST editor, you can filter the displayed selection list according to validity areas: depending on the selection list displayed in each case, you can switch between the following selection lists using the **[right arrow]** and **[left arrow]** keys:
 - All entries
 - Keywords
 - Global declarations
 - Local declarations
 - If you call a function block, a method or a function and enter a left parenthesis for entering the POU parameters, TwinCAT shows a tooltip. This tooltip contains information on the parameters, as they are declared within the POU. The tooltip remains visible until you close it with a mouse click or by placing the focus outside the current view. If you close the tooltip accidentally, you can open it again with **[Ctrl] + [shift key] + [space bar]**.



The pragma attribute 'hide' can be used to exclude variables from **List components** function. (Attribute 'hide' |> 805)

Examples:

Entering a structure variables:

```
1  erg := stVar.
2
3  ▾ bVar1
4  ▾ nVar2
   ▾ nVar3
```

Calling a function block:

```
1  erg := fbInst (
2
3  FUNCTION_BLOCK FB_Sample
4  VAR_INPUT    nVarIn  INT
5  VAR_OUTPUT   nVarOut INT
6
```

Shortcut mode

Shortcut mode enables entering shortcuts for the variable declaration in the declaration editor and in the text editors, in which variable declarations are possible. You can enable this mode by ending a declaration line with the shortcut **[Ctrl] + [Enter]**.

TwinCAT supports the following shortcuts:

- All identifiers apart from the last identifier in a line become variable identifiers of a declaration.
- The data type of the declaration is determined by the last identifier in the line. The following applies:
 - B or BOOL results in BOOL
 - I or INT results in INT
 - R or REAL results in REAL
 - S or STRING results in STRING
- If no data type is specified based on these rules, the data type is automatically set to BOOL, and the last identifier is not used as data type (see example 1).
- Each constant that is entered is interpreted as an initialization or a string length declaration, depending on the type of declaration (see examples 2 and 3).
- An address (such as in %MD12) is automatically extended with the AT attribute (see example 4)
- Text after a semicolon ";" is interpreted as a comment (see example 3).
- All other characters in the line are ignored (see exclamation mark in example 5).

Examples:

Example	Shortcut	resulting declaration
1	bA	bA: BOOL;
2	nA nB I 2	nA, nB: INT := 2;
3	sC S 2; C string	sC: STRING(2); // C string
4	X %MD12 R 5	X AT %MD12: REAL := 5.0;
5	bE !	bE: BOOL;

Smart tag functions



Available from TC3.1 Build 4026

Smart tags facilitate the creation of program code by providing suitable commands for selection directly at the programming element. If you place the cursor on a programming element for which a smart tag function

is available, the symbol  appears. Clicking on  will display the commands you can select. Available Smart Tags:

- For undeclared variables in the implementation part in the ST editor, the smart tag function provides the command **Auto Declare**.

See also:

- [Declaring variables](#) [► 66]
- TC3 User Interface documentation: [Dialog Options - Smart coding](#) [► 981]

7.11 Using pragmas

Pragmas in TwinCAT

A pragma is text in the application source code that is enclosed in curly brackets. Pragmas are used to insert special instructions in the code that the compiler can evaluate. This allows a pragma to influence the properties of one or several variables with respect to precompilation or compilation (code generation). Pragmas unknown to the compiler are read over like a comment.

The instruction string of a pragma can also be multi-line. Please refer to the descriptions of the individual pragmas for details about the syntax.

There are pragmas for different effects: initialization of a variable, monitoring of a variable, forcing of message outputs during the compilation procedure, behavior of a variable under certain conditions, etc.



Case-sensitivity must be respected.

Sample:

```
{warning 'This is not allowed'}
{attribute 'obsolete' := 'datatype FB_Sample not valid!'}
```

Possible insert positions



Pragmas in TwinCAT are not one-to-one implementations of the C preprocessor directives. A pragma has to be positioned like a normal instruction. A pragma cannot be used within an expression.

You can insert a pragma to be evaluated by the compiler at the following positions:

- In the declaration part of a programming block:
 - In the textual declaration editor, you enter pragmas directly as line(s), either at the beginning of the block or before a variable declaration.
 - In the tabular editor, pragmas, which must be above the first declaration line, can be added in the **Attributes** dialog. Double click on the column **Attributes**.
- In a global variable list
- In the implementation part of a programming block:
 - The pragma must be placed at a "instruction position", i.e. at the beginning of a programming block in a separate line, or after a ";" or END_IF, END_WHILE etc..
 - FBD/LD/IL editor: enter pragmas in networks of the FBD/LD/IL editor like a label: To do this, select the command **FBD/LD/IL > Insert label** and then replace the standard text **Label:** in the label text field with the corresponding pragma instruction. If you want to use a pragma in addition to a label, first enter the pragma, then the label.

Incorrect and correct positioning of a conditional pragma:

Incorrect:	Correct:
<pre>{IF defined(abc) } IF x = abc THEN {ELSE} IF x = 12 THEN {END_IF} y := {IF defined(cde) } 12; {ELSE} 13; {END_IF} END_IF</pre>	<pre>{IF defined(abc) } IF x = abc THEN {IF defined(cde) } y := 12; {ELSE} y := 13; {END_IF} END_IF {ELSE} IF x = 12 THEN {IF defined(cde) } y := 12; {ELSE} y := 13; {END_IF} END_IF {END_IF}</pre>



In the POU **Properties**, category **Advanced**, you can enter **Defines**, which can be queried in pragmas.

Scope:

Depending on the type and content of a pragma, it affects the following:

- the following declarations
- specifically the following instruction
- all following instructions until it is canceled with a corresponding pragma.
- all following instructions, until the same pragma is executed with other parameters or the end of the code is reached. "Code" in this context means: declaration part, implementation part, global variable list, type declaration. A pragma, which appears on its own in the first line of the declaration part and is not superseded or canceled by another pragma, is therefore active for the entire object.

Pragma categories

The TwinCAT pragmas are divided into the following categories:

- Attribute pragmas: influence compilation and precompilation ([Attribute pragmas \[► 794\]](#))
- Message pragmas: output of user-defined messages during the compile process ([Message pragmas \[► 793\]](#))
- Conditional pragmas: influence the code generation ([Conditional pragmas \[► 837\]](#))
- User-defined pragmas ([User-defined attributes \[► 794\]](#))

See also:

- [Using the Auto Declare dialog \[► 71\]](#)

7.12 Managing text in a text list

Text lists are used for providing visualization texts in several languages. You can enter the texts in Unicode format, so that all languages and characters are available. Text lists can be exported and imported, so that the texts can be compiled outside the current project.

TwinCAT distinguishes between static text, which is managed in the "GlobalTextList" object, and dynamic text, which is managed in objects of type "Textlist".

Static texts are texts within the visualization, which can only change their language at runtime. The text ID remains fixed.

Dynamic texts can be controlled via an IEC variable containing the text ID. This allows you to display varying text in a visualization element at runtime. For example, a text field can be configured such that it outputs an error text with an error number.

Both types of text list contain a table with text entries. An entry consists of an ID for identification, the source text and its translations. In a text list or a global text list, you can translate a source text into any number of languages. The translations are the basis for language selection and language switching in visualizations.

● Directory for text list files



The directories, which provide text lists for the visualization, are specified in the **Visualization** category of the project properties.

See also:

- [Using the input wizard \[► 135\]](#)
- TC3 User Interface documentation: [Textlist \[► 1045\]](#)

Adding a language and translating text

- ✓ A project with a text list or a global text list is open.
- 1. In the PLC project tree, double-click on an object of type **Textlist** or **GlobalTextList**.
 - ⇒ The **Textlist** menu appears in the menu bar, and the text list opens in the editor.
- 2. Select the command **Add Language** in the **Textlist** menu or the context menu.
- 3. Enter a name for the language, e.g. “en-US”. Quit the dialog with **OK**.
 - ⇒ A column with the heading **en-US** appears.
- 4. Enter the translation of the source text in this column.
 - ⇒ You can add any number of languages.




Whereas the **Default** column has to contain text, you don't have to enter translations in the language column. If no translation is found for a text entry, the default text is displayed automatically.

See also:

- TC3 User Interface documentation: [Command Add Language \[► 1045\]](#)

Exporting a text list

- ✓ A project with a text list or a global text list is open.
- 1. In the PLC project tree, double-click on an object of type **Textlist** or **GlobalTextList**.
 - ⇒ The **Textlist** menu appears in the menu bar, and the text list opens in the editor.
- 2. Select the command **Import/Export Text Lists** in the **Textlist** menu or the context menu.
 - ⇒ The **Import/Export** dialog opens.
- 3. Under **Choose export file** click on , select the directory and enter a file names, e.g. “Text_lists_exported”.
- 4. Enable the option **Export**.
- 5. Quit the **Import/Export** dialog with **OK**.
 - ⇒ TwinCAT exports the text list entries of all text lists within a project to a .csv file. The table contains a column with the text list name.



You can use the **Export All** command to export the text list entries as a .txt file. A file is created for each text list. The directory, in which the export files are stored automatically, can be defined in the project properties.

Example:

Content of the file Text_lists_exported


TextList	Id	Default	en-us
TextList_A	A	Information A	Information A_en
TextList_A	B	Information B	Information B_en: Ok
TextList_A	C	Information C	Information C_en
AlarmGroup	1	Warnung 1	
AlarmGroup	2	Warnung 2	
GlobalTextList		Information G	Information G_en
GlobalTextList		Information H	Information H_en
GlobalTextList		Umschalten	Switch
GlobalTextList		Zähler: %i	Counter: %i

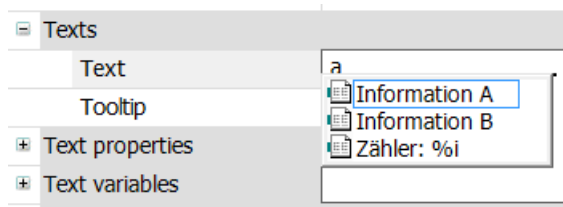
See also:

- TC3 User Interface documentation: [Command Import/Export Text Lists \[► 1046\]](#)

Making the exported file available for the input assistant

✓ You have created a file, for example `Text_lists_exported`, via the command **Import/Export Text Lists**. It contains the texts of the text lists of the project.

1. In the **Tools** menu, click on the command **Options**, category **TwinCAT > PLC Environment > Visualization**, tab **File options**.
 2. In **Text file for textual "IntelliSense"** click on  and select a file, e.g. `Text_lists_exported`. Quit the dialog with **OK**.
- ⇒ If you enter a static text under the property **Texts** for an element in a visualization, TwinCAT offers the source texts contained in the file for selection as soon as you type the first letter.

**See also:**

- [Using the input wizard \[► 135\]](#)

Importing a file with text list entries

An importable file is a file with the format `.csv`. The first line is a header, e.g. "TextList Id Default en_US". The other lines contain text list entries. You obtain such a file when you export the text lists of the project to a file. There you can edit the text list entries outside TwinCAT and then re-import the file. During the import, TwinCAT treats the text list entries for the `GlobalTextList` and for dynamic text lists differently.

`GlobalTextList`:

- If the ID is unknown, TwinCAT does not create and new text list entries.
- TwinCAT ignores modifications that affect the ID or the source text.
- TwinCAT accepts changes to the translations.

`Textlist`:

- For a new ID, TwinCAT complements the corresponding text list with a corresponding text list entry.
- For an existing ID with different source text, the source text in the text list is overwritten with the source text contained in the file.
- TwinCAT accepts changes to the translations.

Import file:

- ✓ A project with a text list or a global text list is open.
1. In the PLC project tree, double-click on an object of type **Textlist** or **GlobalTextList**.
 - ⇒ The **Textlist** menu appears in the menu bar, and the text list opens in the editor.
 2. Select the command **Import/Export Text Lists** in the **Textlist** menu or the context menu.
 - ⇒ The **Import/Export** dialog opens.

3. Under **Choose file to compare or to import**, click on  and select a directory and a file name, e.g. "Text_lists_corrected.csv".
 4. Enable the option **Import**.
 5. Quit the dialog with **OK**.
- ⇒ TwinCAT imports the text list entries from the file into the corresponding text lists.

Example:



Content of the file Text_lists_corrected.csv

TextList	Id	Default	en-us
TextList_A	A	Information A	Information A_en
TextList_A	B	Information B: Ok	Information B_en: Ok
TextList_A	C	Information C	Information C_en
TextList_A	D	Information D	Information D_en
AlarmGroup	1	Warnung 1	Warning 1
AlarmGroup	2	Warnung 2	
GlobalTextList		Information G	Information G_en: Ok
GlobalTextList		Information HH	Information H_en
GlobalTextList		Information I	Information I_en
GlobalTextList		Umschalten	Switch
GlobalTextList		Zähler: %i	Counter: %i

This content is transferred to the corresponding text lists in the project.

TextList	Id	Default	en-us
TextList_A	A	Information A	Information A_en
TextList_A	B	Information B: Ok	Information B_en: Ok
TextList_A	C	Information C	Information C_en
TextList_A	D	Information D	Information D_en
AlarmGroup	1	Warnung 1	Warning 1
AlarmGroup	2	Warnung 2	
GlobalTextList		Information G	Information G_en: Ok
GlobalTextList		Information H	Information H_en
GlobalTextList		Umschalten	Switch
GlobalTextList		Zähler: %i	Counter: %i

Comparing text lists with file and export difference

- ✓ A project with a text list or a global text list is open.
1. In the PLC project tree, double-click on an object of type **TextList** or **GlobalTextList**.
 - ⇒ The **Textlist** menu appears in the menu bar, and the text list opens in the editor.
 2. Select the command **Import/Export Text Lists** in the **Textlist** menu or the context menu.
 - ⇒ The **Import/Export** dialog opens.
 3. Under **Choose file to compare or to import**, click on  and select a directory and a file name for the comparison file, e.g. "Text_lists_corrected.csv".
 4. Under **Choose export file**, click on , select a directory and enter a name for the file that will contain the result of the comparison.
 5. Enable the option **Export only text differences**.
 6. Quit the dialog with **OK**.
- ⇒ TwinCAT reads the import file and compares the text list entries with the same ID. If there are differences, TwinCAT writes the text list entries of the text list into the export file. For the global text list, TwinCAT compares the translations with the source texts. If there are differences, TwinCAT writes the text list entries into the export file.

Example:

TextList	Id	Default	en-us
TextList_A	B	Information B	Information B_en: Ok
AlarmGroup	1	Warnung 1	Warning 1
GlobalTextList		Information G	Information G_en
GlobalTextList		Information H	Information H_en

See also:

- TC3 User Interface documentation: [Command Import/Export Text Lists |► 1046](#)

7.12.1 Managing Static Text in Global Text Lists

The global text list is the central location in the project for texts, which are output in the visualization.

If a text in a visualization element for the first time, TwinCAT automatically creates a global text list. It is added to the PLC project tree as an object and will only exist once. You can open the global text list by double-clicking on the object in an editor.

The global text list contains a table with all static texts you have written in the project visualizations. If you write a further text in a visualization element under the **Texts** property, TwinCAT amends the table automatically. TwinCAT assigns the IDs as consecutive integer numbers, beginning with 0.

You can check, update and synchronize the global text list against the static texts of the visualizations. Here, the options are limited to editing and translating existing text, i.e. it is not possible to write new text. The source text or the ID cannot be edited directly in the table, although a source text can be replaced with another text by creating a replacement file and importing it. Menu commands are available for this purpose.


TwinCAT provides the following commands for consolidating the GlobalTextList:

- Check Visualization Text Ids
- Update Visualization Text Ids
- Remove Unused Text List Records

See also:

- TC3 User Interface documentation: [Textlist \[►_1045\]](#)

Structure of a global text list

Symbol: 

ID	Default	en-us
0	Information G	Information G en
1	Information H	Information H en
2	Umschalten	Switch
3	Zähler: %i	Counter: %i

ID	Unique text identifier
Standard	Source text as string with a maximum of one formatting specification, e.g. information A: %i options. If no translation is written in a language column, TwinCAT uses this text. Double-click in the field to edit the text.
The table contains any number of language columns you may have added. A language column is identified with a language code, which you specified when you created the column with the Add Language command.	
<language code>	Name of the language in the form of a language code, e.g. en-US. This column contains the translation of the text written under Default. If a language code is selected in the visualization manager, a visualization issues the translation during operation. A visualization during operation can switch to another language on request by a user. Double-click in the field to edit the text.

Configuring a visualization element with static text

A text in a GlobalTextList may contain a formatting specification.

- ✓ A project with visualization is open. The **GlobalTextList** object contains the texts that are defined in the visualizations of the project.
1. In the PLC project tree, double-click on the visualization.

- ⇒ The editor opens.
- 2. Select an element that has the **Text** property, for example a text field.
- 3. Enter a text in the **Text** property, for example “Static Information A”.
- ⇒ TwinCAT extends the global text list in the POU view with the new text.

Checking the global text list

- ✓ A project with visualization is open. The **GlobalTextList** object contains the texts that are defined in the visualizations of the project.
- 1. In the PLC project tree, double-click on the **GlobalTextList** object.
 - ⇒ The table containing the static texts as list entries opens.
- 2. In the **Textlist** menu or the context menu of the editor, select the command **Check Visualization Text Ids**.
- ⇒ TwinCAT indicates if a source text from the text list differs from the static text that is identified via the ID. The source text in the global text list does not match the text in the visualizations with the same ID.

See also:

- TC3 User Interface documentation: [Command Check Visualization Text Ids \[► 1047\]](#)

Updating the IDs in the global text list

- ✓ A project with visualization is open. The **GlobalTextList** object contains the texts that are defined in the visualizations of the project.
- 1. In the PLC project tree, double-click on the **GlobalTextList** object.
 - ⇒ The table containing the static texts as list entries opens.
- 2. In the **Textlist** menu or the context menu of the editor, select the command **Update Visualization Text Ids**.
- ⇒ TwinCAT supplements the global text list, if a text in the static text property does not match a source text in the project visualizations.

See also:

- TC3 User Interface documentation: [Command Update Visualization Text Ids \[► 1048\]](#)

Removing IDs from the global text list

- ✓ A project with visualization is open. The **GlobalTextList** object contains the texts that were defined in the visualizations of the project.
- 1. In the PLC project tree, double-click on the **GlobalTextList** object.
 - ⇒ A table with the texts opens.
- 2. In the **Textlist** menu or the context menu of the editor, select the command **Remove Unused Text List Records**.
- ⇒ TwinCAT removes the text list entries, whose IDs are not referenced in the project visualizations.

See also:

- TC3 User Interface documentation: [Command Remove Unused Text List Records \[► 1047\]](#)


Edit global text list with replacement file

A replacement file has the format .csv. The first line is a header: defaultold defaultnew REPLACE. The other lines contain the old source text, the new source text and the command REPLACE. Tab, comma or semicolon is allowed as a separator. A mixture of separators within a file is not allowed.

Sample (tab as separator):

```
defaultalt      defaultneu      REPLACE
Information A   Information A1  REPLACE
```

If you import a replacement file, TwinCAT processes the file line-by-line and implements the specified substitutions in the **GlobalTextList**. In addition, TwinCAT replaces the previous text in the visualizations with the replacement text. If the replacement text was already available as static text, TwinCAT recognizes this, harmonizes the static texts and leaves only one text list entry.

- ✓ A project with a text list or a global text list is open.
- 1. In the PLC project tree, double-click on the **GlobalTextList** object.
 - ⇒ The object opens.
- 2. In the **Textlist** menu or the context menu of the editor, select the command **Import/Export Text Lists**.
 - ⇒ The **Import/Export** dialog opens.
- 3. Under **Choose file to compare or to import**, click on  and select a directory and a file name, e.g. "ReplaceGlobalTextList.csv".
- 4. Enable the option **Import replacement file**.
- 5. Quit the dialog with **OK**.
 - ⇒ The texts in the text lists and the visualizations are replaced.

Sample:

The global text list contains the following source texts:

```
GlobalTextList Counter:%i
GlobalTextList Information A
GlobalTextList Information Aa
GlobalTextList Umschalten
GlobalTextList Zähler:%i
```

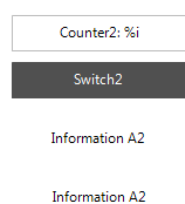
The replacement file contains the following substitutions:

```
defaultalt defaultneu REPLACE
Counter:%i Counter2:%i REPLACE
Information A Information A2 REPLACE
Information Aa Information A2 REPLACE
Umschalten Switch REPLACE
Zähler:%i Counter2:%i REPLACE
```

TwinCAT detects duplicate text list entries and removes one of them. The global text list then contains the following entries:

```
5 Switch2
4 Counter2: %i
3 Information A2
```

The texts in the visualization have been replaced.



See also:

- TC3 User Interface documentation: [Command Import/Export Text Lists](#) [▶ 1046]

7.12.2 Managing Dynamic Text in a Text List

In a text list for dynamic text you can manage, create and translate texts. A text you have written here can be selected in a visualization element in the property **Dynamic texts**. During operation the visualization outputs this text dynamically in the selected language.

You create a text list as an object in the PLC project tree. The text list contains a table with text list entries, which you can edit and expand. An text list entry consists of an ID for identification, the source text and its translations. You can add new text list entries to a text list. Menu commands are available for this purpose.


See also:

- TC3 User Interface documentation: [Textlist \[►_1045\]](#)

Creating an object Text List

1. Select a folder in the **Solution Explorer** in the PLC project tree.
2. In the context menu select the command **Add > Text List...**
 - ⇒ The **Add Text List** dialog opens.
3. Enter a name.
4. Click on **Open**.
 - ⇒ The text list is added to the PLC project tree and opens in the editor.

Structure of a text list

Symbol: 

ID	Default	en-us
A	Information A	Information A_en
B	Information B	Information B_en: OK
C	Information C	Information C_en
D	Information D	Information D_en

ID	Unique text identifier
Standard	Source text as string, e.g. Information A Double-click in the field to edit the text.
The table contains any number of language columns you may have added. A language column is identified with a language code, which you specified when you created the column with the Add Language command.	
<language code>	Name of the language in the form of a language code, e.g. en-US. This column contains the translation of the text written under Default. Provided a language code is selected in the visualization manager, a visualization outputs the translated text during operation. If no translation has been entered, TwinCAT uses the text under Default. During operation a visualization can switch language, is requested by a user.
Empty line	You can edit the line to add your own text.

Creating a text list for dynamic text output

- ✓ A project with visualization is open.
1. In the **Solution Explorer**, select the PLC project object or a folder below it in the PLC project tree.
 2. In the context menu select the command **Add > Text List**.
 3. Enter a name, for example "TextList_A", and quit the dialog with "Open".
 - ⇒ An object of type **Text List** is created.
 4. Click under the **Default** column and open the input field. Enter a text, for example "Information A".
 - ⇒ The source text is created. It serves as a key in the table and as source text for translations.
 5. In column **ID** enter any character string, for example "A".
 6. Double-click in the empty line at the end of the table under Default and enter further text list entries.
 - ⇒ The text list entries with source text and ID are defined. If you configure the property **Dynamic texts** for an element in a visualization, you can now select the text list "TextList_A", for example, and assign the ID "A".

Dynamic text output

In a visualization you can configure the dynamic output of texts that were written in a text list by configuring the property **Dynamic texts** of an element. You can assign a text list and an ID directly, or you can use IEC variables and set the values programmatically.

- ✓ A project with visualization is open, and a text list exists in the PLC project tree.
- 1. Open the text list in the editor, for example "TextList_A".
- 2. Double-click the visualization object.
 - ⇒ The editor opens.
- 3. Drag an element, for example of type **Textfield**, into the visualization.
- 4. Configure its property **Dynamic texts**, by selecting an element from the **Text list** property, for example "TextList_A", and entering an ID from the text list in **Text index**, for example "A". Pay attention to the quotation marks. You can also assign an IEC variable of type STRING for the text list name and the ID.
 - ⇒ The IEC variables enable programmatic access to the texts of the text lists.
- 5. Compile the application, upload it to the controller and start it.
 - ⇒ The visualization outputs the text from the text list in the text field: Information A.

7.13 Using image pools

An image pool is a table of image files. By specifying the ID and the name of the image pool, TwinCAT can unambiguously reference an image file when you use it in the project, for example in a visualization. A project can contain multiple image pools. You can add image pools to the PLC project tree in the Solution Explorer. In a library project, you can assign an image pool as symbol library for the visualization via its object properties.




We recommend reducing the size of an image file as far as possible you integrate it. This optimizes the loading time of the visualization in all visualization variants (TargetVisu, WebVisu and programming system).

When you insert an image element into a visualization and enter an ID (static ID) in the element properties, TwinCAT automatically creates a global image pool. TwinCAT uses the default name "GlobalImagePool".

Note the following points, if the ID of an image file exists in multiple image pools:

- Search order: If you select an image that is managed in the GlobalImagePool, you do not have to specify the name of the image pool. The search order for image files is as follows:
 - 1. GlobalImagePool
 - 2. Image pools assigned to the currently active PLC project
 - 3. Image pools that exist in the **Solution Explorer**, alongside the GlobalImagePool
 - 4. Image pools in libraries
- Unique access: You can address the desired image directly and unambiguously by preceding the ID with the name of the image pool, according to the syntax <collection name>.<image ID>.

7.13.1 Object Image Pool

Symbol: 

The **Image Pool** object contains a table, in which images are assigned IDs.

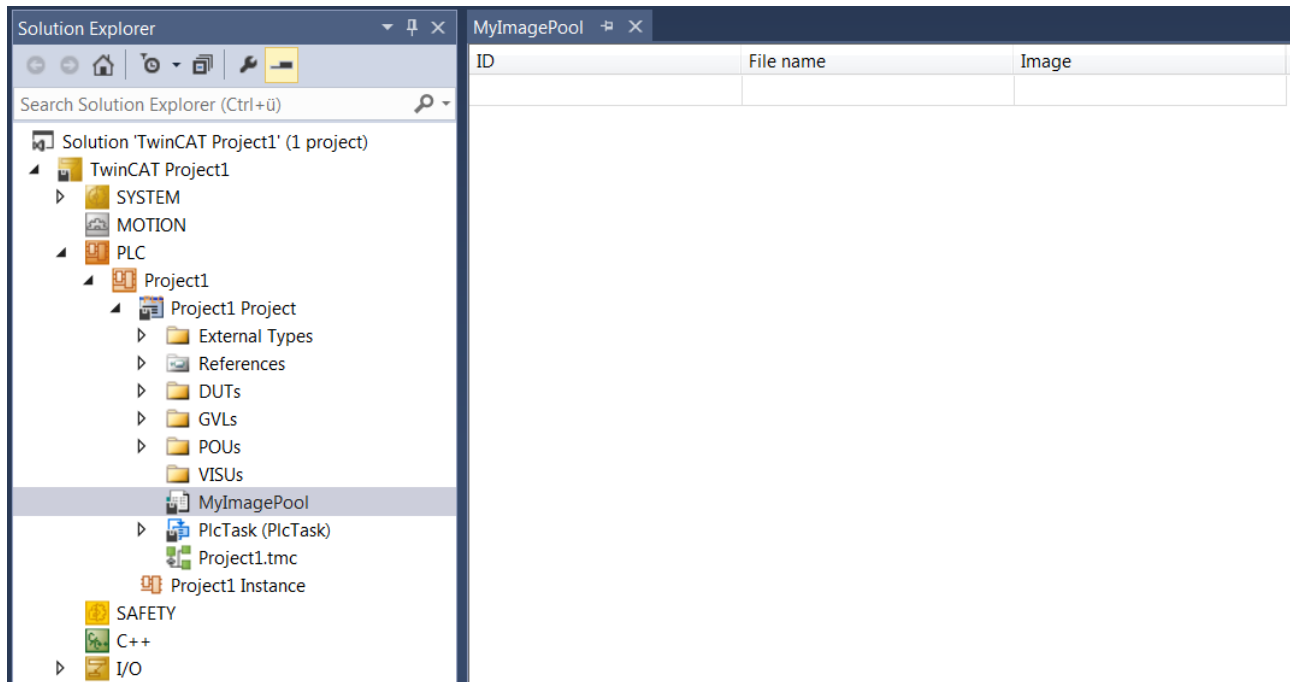
Creating an object Image Pool


1. Select a folder in the **Solution Explorer** in the PLC project tree.
2. In the context menu select the command **Add > Image Pool**.
 - ⇒ The **Add Image Pool** dialog opens.
3. Enter a name.

4. Click on **Open**.

⇒ The image pool is added to the PLC project tree and opens in the editor.


Structure of an image pool



ID	Image ID. This ID is used to reference the image in the visualization, for example.
File name	File path of the image file. When you click the  button, the standard image selection dialog opens.
Image	Shows a thumbnail view of the image.

7.13.2 Creating an Image Pool

Creating an Image Pool

- In the **Solution Explorer**, select the PLC project object or a folder below it in the PLC project tree. In the context menu select the command **Add > Image Pool**.
 - ⇒ The **Add Image Pool** dialog opens.
- Enter a name for the image pool (for example "Images1") and confirm the dialog by clicking the **Open** button.
 - ⇒ The image pool is added in the PLC project tree and opens in an editor.
- Double-click on the **ID** field and assign a suitable ID (for example "Icon1").
- Double-click on the **File name** column. Click on  .
 - ⇒ The standard **Open** dialog opens.
- Select the image file.
- Enter an ID in the **ID** column. If you do not enter an ID, the file name is automatically inserted.
 - ⇒ A thumbnail view of the image file is displayed in the **Image** column. The path is displayed in the **File name**, the name of the file or the ID you specified is displayed in the **ID** field. The image file can now be referenced by the name "Images1.Icon1".

Using an image file in the image visualization element

When you insert an image element in a visualization, you can set the image type:

- Static image: In the element configuration (property **Static ID**) enter the ID of the image file or the name of the image pool + ID. Note the information on search order and access provided above.
- Dynamic image: In the element configuration (property **Bitmap ID variable**) enter the variable that defines the ID of the image file, for example MAIN.imagevar. In online mode a dynamic element can be replaced based on a variable

Using an image file for the visualization background

You can specify an image in the background definition for a visualization. The image can be defined by the name of the image pool plus the file name, as described above for a visualization element.

7.14 Checking the syntax and analyzing the code

TwinCAT offers useful functions to assist with programming and troubleshooting. The syntax check highlights errors during program input and issues corresponding messages.

The static code analysis in TwinCAT provides additional assistance for complying with specified coding guidelines and identifying error-prone constructs.

7.14.1 Checking Syntax

When you enter the code, the TwinCAT precompiler performs some basic checks. Any errors are highlighted with a squiggly red line in the editor.

Once the programming is complete, you have to compile the PLC project. This is done with the **Build** command in the **Project** menu. The compiler checks the program and lists any errors in the **Error List**.

TwinCAT automatically generates the program code from the source code written in the development system before downloading the PLC project to the controller. Before the program code is generated, the system checks the assignments, data types and availability of libraries. In addition, the memory addresses are allocated when the program code is generated.

A check can also be started explicitly via the command **Check all objects** in the **Build** menu. This also detects errors in objects, which are not used in the current PLC project.

TwinCAT lists all errors and warnings in the **Error List**. Double-clicking an error message to open the affected POU. The faulty position is highlighted. Alternatively, you can navigate to the faulty position via the context menu for the error message.

Please also refer to the settings in the TwinCAT options under **Smart Coding**.

See also:

- TC3 User Interface documentation: [Dialog Options - Smart coding \[► 981\]](#)

7.14.2 Code analysis (Static Analysis)

Before a project is loaded onto the target system, TwinCAT 3 PLC uses "Static Code Analysis" to check whether the source code follows specified coding guidelines.

The license-free version of the static code analysis is called "Static Analysis Light". The checks configured in the analysis function are performed automatically whenever code is generated. Further information can be found in section "[Static Analysis Light \[► 150\]](#)".

The licensed version of the static code analysis is the "Static Analysis", which has a greatly extended range of functions and configurations in comparison with the Light version. The static code analysis can be triggered manually or performed automatically during the code generation. Further information on this extension can be found in the documentation "[TE1200 | TC3 PLC Static Analysis](#)".

Static Analysis Light vs. Static Analysis Full

An overview of the different features of the license-free and license-managed variants of Static Analysis is provided below.

Functional aspect	Static Analysis Light (without TE1200 license)	Static Analysis Full (with TE1200 license)
License required	No, usable free of charge	Yes, TE1200 license required
Save/export and load/import (rule) configuration	Not possible, coupled to PLC project properties	Possible (using the Load/Save buttons in the Settings)
Execution is coupled to the compilation process	Yes, not configurable	Configurable (using the Perform static analysis automatically option in the Settings; Manual execution with the help of the command Command 'Run static analysis')
Checking for unused objects (e.g. within a library project)	Not possible	Possible (with the help of the command Command 'Run static analysis [Check all objects]')
Maximum number of reported errors	500 (not configurable) (Further information on the significance of 500 as the maximum number of errors can be found in the Settings)	Configurable (using the setting Maximum number of errors in the Settings)
Maximum number of reported warnings	Output of warnings not possible (see following line)	Configurable (using the setting Maximum number of warnings in the Settings)
Rules: Activation options	<ul style="list-style-type: none"> • Active and output as error • Inactive 	<ul style="list-style-type: none"> • Active and output as error • Active and output as warning • Inactive
Rules: scope	7 coding rules <ul style="list-style-type: none"> • SA0033: Unused variables • SA0028: Overlapping memory areas • SA0006: Write access to multiple tasks • SA0004: Multiple writes access on output • SA0027: Multiple usage of name • SA0167: Report temporary FunctionBlock instances • SA0175: Suspicious operation on string 	More than 100 coding rules
Rules: Precompile wavy underline, QuickFix	Not available	Available
Naming conventions	Not available	Available
Metrics	Not available	Available
Forbidden symbols	Not available	Available

Pragmas and attributes for temporary deactivation of rules	Yes, available in the Light scope: <ul style="list-style-type: none"> • Pragma {analysis ...} • Attribute {attribute 'no-analysis'} • Attribute {attribute 'analysis' := '...'} 	Yes, available in full scope: <ul style="list-style-type: none"> • Pragma {analysis ...} • Attribute {attribute 'no-analysis'} • Attribute {attribute 'analysis' := '...'} • Attribute {attribute 'naming' := '...'} • Attribute {attribute 'nameprefix' := '...'} • Attribute {attribute 'analysis:report-multiple-instance-calls'}
--	--	--

7.14.2.1 Static Analysis Light

Before a project is loaded onto the target system, TwinCAT 3 PLC uses "Static Code Analysis" to check whether the source code follows specified coding guidelines.

The license-free version of the static code analysis is called "Static Analysis Light". The checks configured there are automatically performed following each successful code generation. The required set of rules is defined in the PLC project properties under **Static Analysis**.

Deviations from the rules are issued as error messages in the **Error List**. Each rule has a unique number. If a violation of a rule is detected during the static analysis, the rule number is output in the error list together with an error description based on the following syntax. The abbreviation "SA" stands for "Static Analysis".

Syntax: "SA<rule number>: <rule description>"

Sample for rule number 33 (Unused variables): "SA0033: Not used: Variable 'bSample'"



TwinCAT only analyses the program code of the current project, but not libraries.



Note that the Static Analysis Light is automatically executed following the successful compilation process. If, on the other hand, the code generation was not successful, i.e. if the compiler detected compilation errors, the Static Analysis Light is not executed.



You can use pragmas to disable checks for certain code parts (see below).

Configuring the rule set

The **Static Analysis** category of the PLC project properties defines the checks carried out by the Light version of the static code analysis whenever code is generated.



Scope of the Static Analysis configuration

The parameters you set in the category **Static Analysis** of the PLC project properties are referred to as **Solution options** and therefore affect not only the PLC project whose properties you currently edit. The configured rule set is used for all PLC projects in the development environment.

Available rules within the Static Analysis Light:

- [SA0033: Unused variables](#)
- [SA0028: Overlapping memory areas](#)
- [SA0006: Write access from several tasks](#)
- [SA0004: Multiple writes access on output](#)

- [SA0027: Multiple usage of name](#)
- [SA0167: Report temporary FunctionBlock instances](#)
- [SA0175: Suspicious operation on string](#)

SA0033: Unused variables

Function	Determines variables that are declared but not used within the compiled program code.
Reason	Unused variables make a program less easy to read and maintain. Unused variables occupy unnecessary memory space and take up unnecessary runtime during the initialization.
Importance	medium
PLCopen rule	CP22/CP24

SA0028: Overlapping memory areas

Function	Determines the points due to which two or more variables occupy the same storage space.
Reason	If two variables occupy the same storage space, the code may behave very unexpectedly. This must be avoided in all cases. If the use of a value in different interpretations is unavoidable, for example once as DINT and once as REAL, you should define a UNION. Also, via a pointer you can access a value typed otherwise without converting the value.
Importance	High

Sample:

In the following sample both variables use byte 21, i.e. the memory areas of the variables overlap.

```
PROGRAM MAIN
VAR
  nVar1 AT%QB21 : INT;           // => SA0028
  nVar2 AT%QD5  : DWORD;        // => SA0028
END_VAR
```

SA0006: Write access from several tasks

Function	Determines variables with write access from more than one task.
Reason	A variable that is written in several tasks may change its value unexpectedly under certain circumstances. This can lead to confusing situations. String variables and, on some 32-bit systems, 64-bit integer variables also may even assume an inconsistent state if the variable is written in two tasks at the same time.
Exception	In certain cases it may be necessary for several tasks to write a variable. Make sure, for example through the use of semaphores, that the access does not lead to an inconsistent state.
Importance	High
PLCopen rule	CP10



See also rule SA0103.

i Call corresponds to write access

Please note that calls are interpreted as write access. For example, calling a method for a function block instance is regarded as a write access to the function block instance. A more detailed analysis of accesses and calls is not possible, e.g. due to virtual calls (pointers, interface).

To deactivate rule SA0006 for a variable (e.g. for a function block instance), the following attribute can be inserted above the variable declaration: {attribute 'analysis' := '-6'}

Samples:

The two global variables nVar and bVar are written by two tasks.

Global variable list:

```
VAR_GLOBAL
  nVar   : INT;
  bVar   : BOOL;
END_VAR
```

Program MAIN_Fast, called from the task PlcTaskFast:

```
nVar := nVar + 1;           // => SA0006
bVar := (nVar > 10);       // => SA0006
```

Program MAIN_Slow, called from the task PlcTaskSlow:

```
nVar := nVar + 2;         // => SA0006
bVar := (nVar < -50);     // => SA0006
```

SA0004: Multiple writes access on output

Function	Determines outputs that are written at more than one position.
Reason	The maintainability suffers if an output is written in various places in the code. It is then unclear which write access is actually affecting the process. It is good practice to perform the calculation of the output variables in auxiliary variables and to assign the calculated value to a point at the end of the cycle.
Exception	No error is issued if an output variable is written in different branches of IF or CASE statements.
Importance	High
PLCopen rule	CP12



This rule **cannot** be disabled via a pragma or attribute!
For more information on attributes, see Pragmas and attributes.

Sample:

Global variable list:

```
VAR_GLOBAL
  bVar   AT%QX0.0 : BOOL;
  nSample AT%QW5   : INT;
END_VAR
```

Program MAIN:

```
PROGRAM MAIN
VAR
  nCondition   : INT;
END_VAR

IF nCondition < INT#0 THEN
  bVar := TRUE;           // => SA0004
  nSample := INT#12;      // => SA0004
END_IF

CASE nCondition OF
```

```

INT#1:
    bVar := FALSE;           // => SA0004

INT#2:
    nSample := INT#11;      // => SA0004
ELSE
    bVar := TRUE;           // => SA0004
    nSample := INT#9;       // => SA0004
END_CASE
    
```

SA0027: Multiple usage of name

Function	Determines multiple use of a variable name/identifier or object name (POU) within the scope of a project. The following cases are covered: <ul style="list-style-type: none"> • The name of an enumeration constant is identical to the name in another enumeration within the application or in an included library. • The name of a variable is identical to the name of another object in the application or in an included library. • The name of a variable is identical to the name of an enumeration constant in an enumeration in the application or in an included library. • The name of an object is identical to the name of another object in the application or in an included library.
Reason	Identical names can be confusing when reading the code. They can lead to errors if the wrong object is accessed inadvertently. Therefore, define and follow naming conventions in order to avoid such situations.
Exception	Enumerations declared with the 'qualified_only' attribute are exempt from SA0027 checking because their elements can only be accessed in a qualified manner.
Importance	Medium

Sample:

The following sample generates error/warning SA0027, since the library Tc2_Standard is referenced in the project, which provides the function block TON.

```

PROGRAM MAIN
VAR
    ton : INT;               // => SA0027
END_VAR
    
```

SA0167: Report temporary FunctionBlock instances

Function	Determines function block instances that are declared as temporary variables. This applies to instances that are declared in a method, in a function or as VAR_TEMP, and which are reinitialized in each processing cycle or each function block call.
Reason	Function blocks have a state that is usually retained over several PLC cycles. An instance on the stack exists only for the duration of the function call. It is therefore only rarely useful to create an instance as a temporary variable. Secondly, function block instances are frequently large and require a great deal of space on the stack (which is usually limited on controllers). Thirdly, the initialization and often also the scheduling of the function block can take up quite a lot of time.
Importance	Medium

Samples:

Method FB_Sample.SampleMethod:

```

METHOD SampleMethod : INT
VAR_INPUT
END_VAR
    
```

```
VAR
    fbTrigger : R_TRIG;           // => SA0167
END_VAR
```

Function F_Sample:

```
FUNCTION F_Sample : INT
VAR_INPUT
END_VAR
VAR
    fbSample : FB_Sample;         // => SA0167
END_VAR
```

Program MAIN:

```
PROGRAM MAIN
VAR_TEMP
    fbSample : FB_Sample;         // => SA0167
    nReturn  : INT;
END_VAR

nReturn := F_Sample();
```

SA0175: Suspicious operation on string

Function	Determines code positions that are suspicious for UTF-8 encoding.
Captured constructs	<ol style="list-style-type: none"> 1. Index access to a single-byte string <ul style="list-style-type: none"> ◦ Sample: sVar[2] ◦ Message: Suspicious operation on string: index access '<expression>' 2. Address access to a single-byte string <ul style="list-style-type: none"> ◦ Sample: ADR(sVar) ◦ Message: Suspicious operation on string: Possible index access '<expression>' 3. Call of a string function of the Tc2_Standard library except CONCAT and LEN <ul style="list-style-type: none"> ◦ Sample: FIND(sVar, 'a'); ◦ Message: Suspicious operation on string: Possible index access '<expression>' 4. Single byte literal containing non-ASCII characters <ul style="list-style-type: none"> ◦ Samples: sVar := '99€'; sVar := 'Ä'; ◦ Message: Suspicious operation on string: literal '<literal>' contains non-ASCII characters
Importance	Medium

Samples:

```
VAR
    sVar : STRING;
    pVar : POINTER TO STRING;
    nVar : INT;
END_VAR

// 1) SA0175: Suspicious operation on string: Index access
sVar[2]; // => SA0175

// 2) SA0175: Suspicious operation on string: Possible index access
pVar := ADR(sVar); // => SA0175

// 3) SA0175: Suspicious operation on string: Possible index access
nVar := FIND(sVar, 'a'); // => SA0175

// 4) SA0175: Suspicious operation on string: Literal '<...>' contains Non-ASCII character
sVar := '99€'; // => SA0175
sVar := 'Ä'; // => SA0175
```

Pragmas and attributes for Static Analysis Light

A pragma or an attribute can be used to exclude certain parts of the code from the check. Use the `pragma {analysis ...}` [▶ 155], to turn off coding rules in the implementation part and the `attribute {attribute 'analysis' := '...'} [▶ 155]`, to turn off coding rules in the declaration part.

Requirement: You have activated the rules in the project properties.

In addition, you can use the `attribute {attribute 'no-analysis'} [▶ 156]` to exclude programming objects from the static analysis.



Rules that are disabled in the project properties cannot be activated by a pragma or attribute.



Rule SA0004 (Multiple writes access on output) cannot be disabled by a pragma.

Pragma {analysis ...}

You can use the pragma `{analysis -/+<rule number>}` in the implementation part of a programming block in order to disregard individual coding rules for the following code lines. Coding rules are deactivated by specifying the rule numbers preceded by a minus sign ("-"). For activation they are preceded by a plus sign ("+"). You can specify any number of rules in the pragma with the help of comma separation.

Insertion location:

- Deactivation of rules: In the implementation part of the first code line from which the code analysis is disabled with `{analysis - ...}`.
- Activation of rules: After the last line of the deactivation with `{analysis + ...}`.
- For rule SA0164, the pragma can also be inserted in the declaration part before a comment.

Syntax:

- Deactivation of rules:
 - one rule: `{analysis -<rule number>}`
 - several rules: `{analysis -<rule number>, -<further rule number>, -<further rule number>}`
- Activation of rules:
 - one rule: `{analysis +<rule number>}`
 - several rules: `{analysis +<rule number>, +<further rule number>, +<further rule number>}`

Samples:

Rule 24 (only typed literals permitted) is to be disabled for one line (i.e. in these lines it is not necessary to write "nTest := DINT#99") and then enabled again:

```
{analysis -24}
nTest := 99;
{analysis +24}
nVar := INT#2;
```

Specification of several rules:

```
{analysis -10, -24, -18}
```

Attribute {attribute 'analysis' := '...'} [▶ 155]

You can use the attribute `{attribute 'analysis' := '-<rule number>'}` to switch off certain rules for individual declarations or for a complete programming object. The code rule is deactivated by specifying the rule number(s) with a minus sign in front. You can specify any number of rules in the attribute.

Insertion location:

above the declaration of a programming object or in the line above a variable declaration

Syntax:

- one rule: {attribute 'analysis' := '-<rule number>'}
- several rules: {attribute 'analysis' := '-<rule number>, -<further rule number>, -<further rule number>'}

Samples:

Rule 33 (unused variables) is to be disabled for all variables of the structure.

```
{attribute 'analysis' := '-33'}
TYPE ST_Sample :
STRUCT
  bMember : BOOL;
  nMember : INT;
END_STRUCT
END_TYPE
```

Checking of rules 28 (overlapping memory areas) and 33 (unused variables) is to be disabled for variable nVar1.

```
PROGRAM MAIN
VAR
  {attribute 'analysis' := '-28, -33'}
  nVar1 AT%QB21 : INT;
  nVar2 AT%QD5 : DWORD;

  nVar3 AT%QB41 : INT;
  nVar4 AT%QD10 : DWORD;
END_VAR
```

Rule 6 (concurrent access) is to be disabled for a global variable, so that no error message is generated if write access to the variable occurs from more than one task.

```
VAR_GLOBAL
  {attribute 'analysis' := '-6'}
  nVar : INT;
  bVar : BOOL;
END_VAR
```

Attribute {attribute 'no-analysis'}

You can use the {attribute 'no-analysis'} attribute to exclude an entire programming object from the static analysis check. For this programming object no checks are carried out for the coding rules, naming conventions and forbidden symbols.

Insertion location:

above the declaration of a programming object

Syntax:

```
{attribute 'no-analysis'}
```

Samples:

```
{attribute 'qualified_only'}
{attribute 'no-analysis'}
VAR_GLOBAL
  ...
END_VAR

{attribute 'no-analysis'}
PROGRAM MAIN
VAR
  ...
END_VAR
```


7.15 Orientation and navigation

7.15.1 Using the Cross Reference List to find Occurrences

You can have the locations of a variable or a POU (program, function block, function) output in a so-called **Cross Reference List**, from where you can jump to the respective locations in the project.

✓ A POU is open in the editor.

1. Place the cursor on a variable or the name of a POU in the declaration or in the implementation.
2. In the menu **PLC > Window** select the command **Cross Reference List**, or select the command **Find All References** in the context menu of the editor.

⇒ The **Cross Reference List** view opens and shows the locations the variables or the POU. It always shows the declaration location and indented below it the locations where it is used in the project. If you search for a structured variable or a function block name, the locations of the members or the function block instances are shown indented below the declaration location.

3. Double-click on a location of the cross reference list.

⇒ The corresponding object opens in the editor, and the location is highlighted.

If the **Cross Reference List** view is open, you can also specify a variable, POU or DUT as follows to find its locations:

- In the menu **Tools > Options** under **TwinCAT > PLC Environment > Smart Coding**, enable the option **Automatically list selection in cross reference view**. Then position the cursor on the name of the variable/POU/DUT in the editor window the POU.
- Manually enter the variable name in the **Name** field.



You can use the placeholders “*” (any number of characters) or “?” (exactly one character) in combination with a substring of a variable identifier.

See also:

- TC3 User Interface documentation: [Command Find all references \[▶ 881\]](#)
- TC3 User Interface documentation: [Command Cross Reference List \[▶ 940\]](#)
- TC3 User Interface documentation: [Dialog Options - Smart coding \[▶ 981\]](#)

7.15.2 Finding Declarations

TwinCAT offers the option of searching the entire project for the definition location of a variable or function. The function block containing the definition opens in the editor, and the declaration is selected.

Finding the declaration of a variable

✓ A POU is open in the editor.

1. Place the cursor on an identifier in the implementation.
2. In the context menu of the editor select the command **Go To Definition**.

⇒ The POU containing the definition opens in the editor, and the variable definition is selected. If the definition is located in a “compiled” library, the corresponding function block opens in the library manager.



You can use the command in offline and online mode.

Examples:

The following function block contains a function block definition (fbInst), a program call (SampleProg()) and a function block call (fbInst.nOut):

```

VAR
    fbInst : FB_Sample;
    nVar : INT;
    nRes : INT;
END_VAR

SampleProg();
nVar := SampleProg.nVar1;
nRes := fbInst.nOut;

```

If you place the cursor on SampleProg, the command opens the SampleProg program in its editor.

If you place your cursor on fbInst, the command puts the focus on the declaration window in line fbInst : FBSample;

If you place your cursor on nOut, the command opens the function block FB_Sample in its editor.

See also:

- TC3 User Interface documentation: [Command Go To Definition \[► 880\]](#)

7.15.3 Set and use bookmarks

You can use bookmarks to facilitate navigation in long programs. Bookmarks can be used in all programming language editors, except SFC (Sequential Function Chart). Commands can be used to navigate directly to the selected program locations.



By default, the bookmarks are stored in the user options file of the Visual Studio project (.suo). This file is user-specific and therefore not compatible with a source control management system. Accordingly, the bookmarks are only valid for the local project.

In order to save the bookmarks across users, there is the option of additionally saving them in a separate file ([Write Bookmarks to File \[► 920\]](#)), which is then part of the TwinCAT archive options. This file is also not compatible with a source control management system.


Set/clear bookmarks


✓ A programming block is open in the editor.

1. Place the cursor in any program line.
2. From the menu **PLC > PLC Bookmark**, select the command **Enable/disable bookmarks**.

⇒ A bookmark is set at this point in the program. This is indicated by the bookmark icon .

3. Set several bookmarks at different points in the program.
4. To delete a bookmark, place the cursor in a program line with a bookmark.
5. From the menu **PLC > PLC Bookmark**, select the command **Enable/disable bookmarks**.

⇒ The bookmark is removed again. The bookmark icon  is deleted. Alternatively, you can delete one or more bookmarks in the **PLC Bookmarks** window using the button

 For this, the corresponding bookmarks must be selected.



Select in the menu **PLC > PLC Bookmark** the command **Delete all bookmarks (active editor)** to remove all bookmarks of the active programming block.

Select in the menu **PLC > PLC Bookmarks** the command **Delete all bookmarks** to delete all bookmarks of a project.

See also:

- [Command Enable/disable bookmarks \[► 953\]](#)
- [Command Clear All Bookmarks \(active editor\) \[► 955\]](#)
- [Command Clear All Bookmarks \[► 954\]](#)



Jump to bookmarks within a programming block

- ✓ A programming block is open in the editor. Several bookmarks are set.
- 1. From the menu **PLC > PLC Bookmark**, select the command **Next Bookmark (active editor)**.
 - ⇒ Depending on the current cursor position, the cursor jumps down to the next bookmark.
- 2. From the menu **PLC > PLC Bookmark**, select the command **Previous Bookmark (active editor)**.
 - ⇒ Depending on the current cursor position, the cursor jumps up to the next bookmark.

See also:

- [Command Next Bookmark \(active editor\) \[► 954\]](#)
- [Command Previous Bookmark \(active editor\) \[► 954\]](#)

Jumping to bookmarks of different programming blocks of a project

- ✓ A project with several programming blocks is open. Several bookmarks are set in different programming blocks.
- 1. From the menu **PLC > Window** select the command **PLC Bookmarks**
 - ⇒ The **Bookmarks** view opens.
All bookmarks of the project are listed in the view in tabular form.
- 2. Click the **Next Bookmark**  button.
 - ⇒ In the **Bookmark** view the bookmark in the row below the currently selected bookmark is selected.
The programming block with the newly selected bookmark of the table opens in the editor and the row with the bookmark is selected.
- 3. According to step 2 you can use the button **Previous Bookmark**  to jump to the bookmark of the project that is displayed in the **Bookmark** view in the row above.

See also:

- [Command PLC Bookmarks \[► 950\]](#)
- [Command Next Bookmark \[► 953\]](#)
- [Command Previous Bookmark \[► 954\]](#)

7.16 Find and Replace in the entire project

In TwinCAT you can search for strings in individual objects or the entire project and replace them with another string, as required.

1. In the menu **Edit > Find and Replace** enable the command **Quick Find**.
 - ⇒ The **Find and Replace** dialog opens.
2. Enter the required string in the **Find what** field.
3. In the **Find in** selection list, specify the objects to be searched.
4. Select the **search options**.
5. Select the **result options**.
6. Click **Find Next**.
 - ⇒ The first hit is displayed in the editor.
7. Click **Find all** to obtain an overview of all hits.
 - ⇒ The **Search results** screen opens, showing a list of the hits.
8. Click on **Replace in files** to replace the search string with another string.
9. In the field **Replace with** enter the string to replace the original string.
10. Click **Replace**.
11. Click **Replace All** to replace all strings.

See also:

- TC3 User Interface documentation: [Command Quick Replace \[► 884\]](#)
- TC3 User Interface documentation: [Command Quick Find \[► 882\]](#)

7.17 Refactoring

In general, refactoring is a method to improve the design of existing software, without changing its behavior.

In TwinCAT, refactoring offers functionalities for renaming object and variable names and for updating function block connections. You can display all locations of renamed objects and variables show and then rename them individually or collectively. In addition, in the TwinCAT options (**Tools > Options**) under category **TwinCAT > PLC Environment > Refactoring** you can configure whether and where TwinCAT automatically issues refactoring prompts.

Renaming a global variable

Renaming a global variable across the project:

- ✓ You have opened a project containing at least one function block `FB_Sample` and a global variable list `GVL`. The global variable list is open in your editor and contains the declaration of a variable, for example `nGlob1`. `FB_Sample` uses `nGlob1`.
 1. Select the name of a global variable, for example “`nGlob1`”.
 2. In the context menu select the command **Refactoring > Rename 'nGlob1'**.
 3. In the dialog **Rename** enter a new name in the input field **New name**, for example “`nGlobNew`”, and click **OK**.
 - ⇒ The **Refactoring** dialog opens. In the project tree on the left the objects `GVL` and `FB_Sample` are shown in red with a yellow background. In the window on the right, `FB_Sample` is open in its editor, and `nGlob1` is already renamed to `nGlobNew`.
 4. Click on **OK**.
 - ⇒ In your project all instances of the global variable `nGlob1` have been renamed to `nGlobNew`.

Renaming a global variable across the project with the exception of a POU:

1. Select the name of a global variable, for example “`nGlob1`”.
2. In the context menu select the command **Refactoring > Rename 'nGlob1'**.
3. In the dialog **Rename** enter a new name in the input field **New name**, for example “`nGlobNew`”, and click **OK**.
 - ⇒ The **Refactoring** dialog opens. In the project tree on the left the objects `GVL` and `FB_Sample` are shown in red with a yellow background. In the right window, the function block `FB_Sample` is open in its editor. Instead of `nGlob1`, `nGlobNew` is listed.
4. Position the cursor in the window on the right and open the context menu.
5. Select the command **Reject this object** and click **OK**.
 - ⇒ Your project contains the global variable `nGlob1` in `FB_Sample`. The variable `nGlobNew` is specified in the objects in which the variable occurred otherwise. The message window shows an error message, indicating that the identifier `nGlob1` is not defined.

See also:

- TC3 User Interface documentation: [Command Rename '<variable>' \[► 886\]](#)

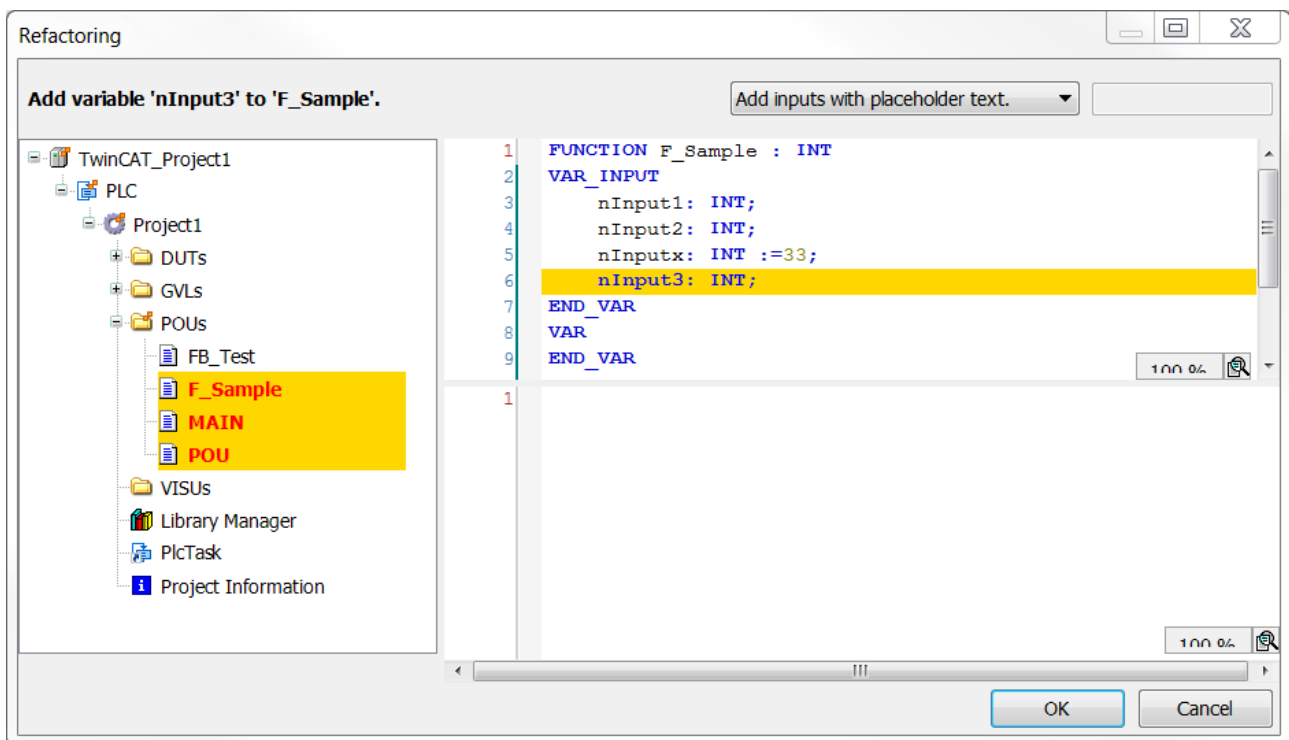
Adding and removing input variables

In the declaration part of function blocks for refactoring commands, you can add or remove input or output variables. TwinCAT updates the locations where the function blocks are called/used accordingly; you can accept or discard the update for each location. The **Refactoring** dialog is used for this purpose.

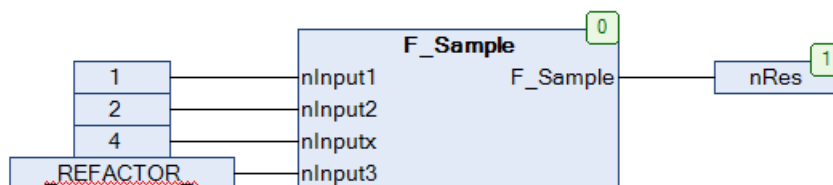
- ✓ You have opened a function `F_Sample` in the editor. The function already has input variables `nInput1`, `nInput2` and `nInputx`. It is called in the programs `MAIN` and `POU`.
 1. Put the focus in the declaration part of the function `F_Sample`.
 2. In the context menu select the command **Refactoring > Add variable**.

- ⇒ The standard dialog for declaring a variable appears.
- 3. Declare the variable nInput3 with scope VAR_INPUT and data type INT. Close the dialog with **OK**.
 - ⇒ The **Refactoring** dialog appears (see figure below). The affected areas are marked in yellow.
- 4. At the top right select the option **Add inputs with placeholder text**.
- 5. In the window on the left, place the cursor on one of the objects highlighted in yellow, for example MAIN. In the context menu select the command **Accept whole project**, in order to add the new variable at the locations where F_Sample is used in the whole project.
 - ⇒ The window on the right shows the change in the implementation part of MAIN: Placeholder `_REFACTOR_` appears where the new variable was inserted.
- 6. Close the **Refactoring** dialog with **OK**.
- ⇒ Select the command **Edit > Find and Replace**. Find “_REFACTOR_” in the project, in order to check the affected locations and edit them as required.

i Alternatively, you can add the new variable directly with an initialization value of your choice, without using a placeholder to start with. In this case, select the option “Add inputs with the following value” under step 4 and enter the value to the right.



Example for a new variable with placeholder text in a CFC function block:



Please note that refactoring can also be used for removing variables.

See also:

- TC3 User Interface documentation: [Command Remove '<variable>' \[▶ 889\]](#)
- TC3 User Interface documentation: [Command Add '<variable>' \[▶ 887\]](#)

Rearranging variables in the declaration

You can use refactoring to change the order of declarations in the declaration part of function blocks. This is possible for declarations the validity ranges VAR_INPUT, VAR_OUTPUT or VAR_IN_OUT.

```
VAR_INPUT
  nInVar2 : INT;
  nInVar1 : INT;
  in      : DUT;
  bVar    : BOOL;
  nInVar3 : INT;
END_VAR
```

- ✓ You have opened the declaration part of a POU, which may contain the declarations shown above, for example.
 1. Place the cursor in this declaration block and press the right mouse button to open the context menu.
 2. In the context menu select the command **Refactoring > Reorder variables**.
 - ⇒ The **Reorder** dialog appears with a list of the VAR_INPUT variables.
 3. For example, select the entry `nInVar1 : INT;` and use the mouse to drag it before the entry `nInVar2 : INT;`.
 - ⇒ The declaration of `nInVar1` is now at the top.
 4. Close the dialog with **OK**.
 - ⇒ The **Refactoring** dialog appears. The affected areas are marked in yellow (see figure at the top).
 5. Confirm with **OK**.
 - ⇒ The new order is transferred to the function block.

See also:

- TC3 User Interface documentation: [Command Reorder variables \[▶ 889\]](#)

Changing a variable declaration and applying refactoring automatically

Refactoring provides support for renaming variables in the declaration (using the autodeclaration feature).

- ✓ You have opened a function block `FB_Sample` in the editor. The function block has an input variable “nVAR”.
- ✓ In the TwinCAT options (**Tools > Options**) in the **TwinCAT > PLC Environment > Refactoring** category, the option **On renaming variables** is activated.
 1. Select the variable `nVar` in the declaration of `FB_Sample`. Alternatively, you can place the cursor before or in the variable.
 2. Select the **Auto Declare** command in the **Edit** menu or in the context menu.
 - ⇒ The **Auto Declare** dialog opens. The dialog contains the settings of “nVar”.
 3. Change the name from “nVar” to “nCounterVar”.
 4. The **Apply changes using refactoring** option appears and is enabled. When changing the variable name, this option appears independently of the settings in the TwinCAT options. However, it is only activated automatically if the TwinCAT refactoring option mentioned in the prerequisites is activated.
 5. Click on **OK**.
 - ⇒ The dialog **Refactoring** opens. All locations affected by the variable renaming are marked there and can be changed.

See also:

- TC3 User Interface documentation: [Command Auto Declare \[▶ 875\]](#)
- TC3 User Interface documentation: [Dialog Options - Refactoring \[▶ 979\]](#)

7.18 Data persistence

Persistent data retain their values when the PLC project is reloaded. The values of persistent data can be restored after an uncontrolled termination (power failure), a download or a cold start.

In addition to the PERSISTENT variables, the RETAIN variables are a further way of keeping data remanent. RETAIN variables likewise retain their values in the case of uncontrolled termination (power failure), a download or a cold start.

See also:

- [Resetting the PLC project \[► 218\]](#)
- [Reference Programming: Remanent variables – PERSISTENT, RETAIN \[► 691\]](#)
- [Retain data](#)

7.19 Using function blocks for implicit checks

TwinCAT provides special POUs, which implement implicit monitoring functions. You can add these special POUs to an application to make implicitly provided monitoring functionality available. At runtime these functions check the bounds of arrays or subrange types, the validity of pointer addresses, or division by 0. Note that this capability may be disabled on devices if such test function blocks are provided by a specific implicit library.

7.19.1 Object POUs for implicit checks

Object Create POU for implicit checks

1. Select a folder in the PLC project tree.
2. In the context menu select the command **Add > POU for implicit checks...**
 - ⇒ The dialog **Add POU for implicit checks** opens.
3. Activate the desired functions.
4. Click on the **Open** button.
 - ⇒ The selected POUs are inserted in the PLC project tree.
5. Open the POUs in the editor.
6. Adapt the implementation proposal to your requirements.

In order to prevent multiple integration, a monitoring function that may already have been integrated is no longer available for selection in the **Add POU for implicit checks** dialog.

● Do not change the declaration part



To maintain the functionality of the monitoring functions, the declaration part must not be modified. The only exception is to add local variables.

● No online change possible



After removing implicit monitoring functions, such as CheckBounds, from the project, online change is no longer possible, only a download. A corresponding message is issued.

● Implicit checks for function blocks from libraries



TwinCAT does not perform implicit checks on function blocks from libraries. However, you can use the compiler definition "checks_in_libs" to extend the check to function blocks from libraries. To do this, enter the compiler definition "checks_in_libs" from the **PLC project properties** in the **Compiler defines** field, which you can find in the **Compile** category. This affects source libraries (*.library).

As of TC3.1 Build 4026, the check can also be extended to function blocks from protected libraries (*.compiled-libraries). To do this, you must activate the "Allow implicit checks for compiled libraries" option in the **Common** category in the **PLC project properties** before saving and installing it as a library. In addition, you must also enter the compiler definition "checks_in_libs" for protected libraries.

● Excluding individual POU's from the check



You can disable the checking of special POU's in the project with the `attribute 'no_check'` [► 812].

Dialog Add POU for implicit checks

Available functions

Monitoring function	Type
CheckBounds	Bound checks: Appropriate treatment of field bound violations (e.g. by setting an error flag or by changing the field index).
CheckDivDInt	Division checks: Monitoring of the divisor value, to avoid division by 0.
CheckDivLInt	
CheckDivReal	
CheckDivLReal	
CheckRangeSigned	Range checks: Monitoring of the range bounds of a subrange type at runtime. Applies to the data types DINT/UDINT
CheckRangeUnsigned	
CheckLRangeSigned	L range checks: Monitoring of the range bounds of a subrange type at runtime. Applies to the data types LINT/ULINT
CheckLRangeUnsigned	
CheckPointer	Pointer checks: For this function you have to complete the entire implementation code yourself. See the help page for POU CheckPointer. The purpose of the function is to monitor whether the transferred pointer points to a valid memory address, and whether the orientation of the referenced memory area matches the type of variable to which the pointer points. If both conditions are met, the pointer itself is returned. Otherwise, CheckPointer should perform appropriate troubleshooting. In the same way, CheckPointer also monitors variables of type REFERENCE TO

See also:

- Reference Programming: `Attribute 'no_check'` [► 812]

7.19.1.1 Bound Checks (POU CheckBounds)

Functions for checking the field boundaries: CheckBounds

The purpose of this monitoring function is to deal with field bound violations as appropriate. A response to a violation may be setting of an error flag or changing the array index, for example. The check is only performed with a variable array index. A consistently faulty array index results in a compiler error. TwinCAT calls the function implicitly as soon as an ARRAY variable is assigned values.

After inserting the function, automatically generated code appears in the declaration part and the implementation part.



● Do not change the declaration part

To maintain the functionality of the monitoring functions, the declaration part must not be modified. The only exception is to add local variables.

Standard implementation

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
{noflow}
IF index < lower THEN
    CheckBounds := lower;
ELSIF index > upper THEN
    CheckBounds := upper;
ELSE
    CheckBounds := index;
END_IF
{flow}
```

When the function is called it is assigned the following input parameters:

- index: Index of the array element
- lower: Lower bound of the array range
- upper: Upper bound of the array range

The return value is the index of the array element, provided it is in the valid range. Otherwise TwinCAT returns the upper or lower bound, depending on which bound was exceeded.

Sample implementation**Declaration part:**

```
// Implicitly generated code : DO NOT EDIT
FUNCTION CheckBounds : DINT
VAR_INPUT
    index, lower, upper: DINT;
END_VAR
// User defined local variables
VAR
    sMessageLow : STRING := 'CheckBounds: Index too low (%d)';
    sMessageHigh : STRING := 'CheckBounds: Index too high (%d)';
END_VAR
```

Implementation part:

```
{noflow}
// Index too low
IF index < lower THEN
    CheckBounds := lower;
    // Increase global counter
    GVL_CheckBounds.nTooLow := GVL_CheckBounds.nTooLow + 1;
    // Log message
    ADSLOGDINT(msgCtrlMask := ADSLOG_MSGTYPE_WARN,
               msgFmtStr := sMessageLow,
               dintArg := index);

// Index too high
ELSIF index > upper THEN
    CheckBounds := upper;
    // Increase global counter
    GVL_CheckBounds.nTooHigh := GVL_CheckBounds.nTooHigh + 1;
    // Log message
    ADSLOGDINT(msgCtrlMask := ADSLOG_MSGTYPE_WARN,
               msgFmtStr := sMessageHigh,
               dintArg := index);

// Index OK
ELSE
    CheckBounds := index;
END_IF
{flow}
```

Like in the standard implementation, in this sample implementation a violation of the array range is corrected by returning the upper or lower bound. In addition a global counter is incremented, if the array is accessed outside the defined array range. The global counters therefore represent the number of upper or lower bound violations for entire project. In addition, the function ADSLOGDINT from the Tc2_System library is used to issue a violation of the array bounds as a warning in the message window.

Application sample

In the program listed below, the index exceeds the defined upper bound of the field aSample. The CheckBounds function corrects this access to the array, which takes place outside the defined array bounds.

```
PROGRAM MAIN
VAR
  aSample : ARRAY[0..7] OF BOOL;
  nIndex  : INT := 10;
END_VAR
aSample[nIndex] := TRUE;
```

In this example, the CheckBounds function causes index 10 to be changed to the upper bound of the array range of aSample (7). The element aSample[7] is assigned the value TRUE. In this way, the function corrects array accesses outside the valid field range. However, it is essential that the cause of the error, namely the access to the element in position 10, is also corrected. So that this automatically corrected faulty access does not remain undiscovered, a warning is displayed in the message window in this example, as described above.

See also:

- [Using function blocks for implicit checks \[► 163\]](#)
- [Object POUs for implicit checks \[► 163\]](#)

7.19.1.2 Division checks (POUs CheckDivDInt, CheckDivLInt, CheckDivReal, CheckDivLReal)

Functions for avoiding the divisor value "0": CheckDivDInt, CheckDivLInt, CheckDivReal and CheckDivLReal

The functions CheckDivDInt, CheckDivLInt, CheckDivReal and CheckDivLReal can be used to avoid division by 0. If you integrate these functions in the PLC project, they are called before each division that occurs in the code.



Do not change the declaration part

To maintain the functionality of the monitoring functions, the declaration part must not be modified. The only exception is to add local variables.

Standard 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 Implementation suggestion
{noflow}
IF divisor = 0 THEN
  CheckDivReal := 1;
ELSE
  CheckDivReal := divisor;
END_IF
{flow}
```

The DIV operator uses the output from the function CheckDivReal as divisor. In the sample program below, division by 0 is avoided by changing the value of the divisor "fDivisor", with was implicitly initiated with "0", to "1" via the function CheckDivReal before the division is executed. Thus the result of the division is 799.

```
PROGRAM MAIN
VAR
  fResult   : REAL;
  fDivident : REAL := 799;
  fDivisor  : REAL := 0;
END_VAR
fResult := fDivident / fDivisor;
```

7.19.1.3 Range/LRange Checks (POUs CheckRangeSigned, CheckRangeUnsigned, CheckLRangeSigned, CheckLRangeUnsigned)

Functions for monitoring the range bounds of a subrange type

- CheckRangeSigned checks subrange types of the type SINT, INT, DINT.
- CheckRangeUnsigned checks subrange types of the type BYTE, WORD, DWORD, USINT, UINT, UDINT.
- CheckLRangeSigned checks subrange types of the type LINT.
- CheckLRangeUnsigned checks subrange types of the type LWORD, ULINT.

The purpose of these monitoring functions is to deal with range bound violations as appropriate. A response to a violation may be setting of an error flag or changing a value, for example. The functions are called implicitly when a variable is assigned a value of the subrange type.

● Do not change the declaration part

i To maintain the functionality of the monitoring functions, the declaration part must not be modified. The only exception is to add local variables.

When the function is called, the following input parameters are transferred to it:

- value: Value to be assigned to the variables of the subrange type.
- lower: Lower range bound
- upper: Upper range bound

The return value is the assigned value itself, provided it is in the valid range. Otherwise the upper or lower bound is returned, depending on the violation of the lower bound.

The assignment `i := 10*y` is now implicitly replaced by
`i := CheckRangeSigned(10*y, -4095, 4095);`

If `y` has the value "1000", for example, the variable `i` is not assigned the value "10*1000=10000", as specified in the original code, but the value of the upper range bound, i.e. "4095".

The same applies to the functions CheckRangeUnsigned, CheckLRangeSigned and CheckLRangeUnsigned.

i If no function for range monitoring is available, no subrange validation is performed during runtime with corresponding variables! In this case you can assign a variable of subrange type DINT/UDINT any value between -2147483648 and +2147483648 or between 0 and 4294967295. A variable of subrange type LINT/ULINT can then be assigned any value between -- 9223372036854775808 and +9223372036854775807 or between 0 and 18446744073709551615.

● Infinite loops

i When you integrate field monitoring functions, infinite loops can occur. This is the case, for example, if the counter variable of a FOR loop is of the subrange type and the loop range leaves the defined subrange.

Example for an infinite loop:

```
VAR
  nVar : DINT(0..10000);
  ...
END_VAR
FOR nVar := 0 TO 10000 DO
  ...
END_FOR
```

The program never leaves the FOR loop, since the monitoring function CheckRangeSigned prevents that `nVar` is set to a value greater than 10000.

Standard implementation of the function CheckRangeSigned

If a DINT variable of a signed subrange type is assigned a value, this necessitates an automatic call of the function CheckRangeSigned. The function, which limits the assignment to the subrange specified in the variable declaration, is implemented in ST as follows by default:

Declaration part:

```
// Implicitly generated code : DO NOT EDIT
FUNCTION CheckRangeSigned : DINT
VAR_INPUT
    value, lower, upper : DINT;
END_VAR
```

Implementation:

```
// Implicitly generated code : Only an Implementation suggestion
{noflow}
IF (value < lower) THEN
    CheckRangeSigned := lower;
ELSEIF (value > upper) THEN
    CheckRangeSigned := upper;
ELSE
    CheckRangeSigned := value;
END_IF
{flow}
```


7.19.1.4 Pointer Checks (POU CheckPointer)

Monitoring function CheckPointer for pointers


Use this function to monitor the memory access of pointers during runtime. In contrast to other monitoring functions, there is no default implementation for CheckPointer. The user must carry out the implementation himself!

The purpose of the CheckPointer function is to check whether the transferred pointer points to a valid memory address, and whether the orientation of the referenced memory area matches the type of variable to which the pointer points. If both conditions are met, the pointer itself is returned. Otherwise, the function should perform proper error handling.

Do not change the declaration part

 To maintain the functionality of the monitoring functions, the declaration part must not be modified. The only exception is to add local variables.

 The monitoring function is not called implicitly for the THIS pointer and the SUPER pointer.

 The function CheckPointer also acts on variables of the type REFERENCE in the same way as on pointer variables.

Template

Declaration:

```
// 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 (incomplete!):

```
// No standard way of implementation. Fill your own code here
{noflow}
CheckPointer := ptToTest;
{flow}
```

When the function is called, TwinCAT transfers the following input parameters to it:

- ptToTest: Target address of the pointer
- iSize: Size of the referenced variable in bytes
- iGran: Granularity of the referenced variable in bytes. i.e. the largest non-structured data type contained in the referenced variables.
- bWrite: Type of access (TRUE = write access, FALSE = read access)

If the check result is positive, the input pointer is returned unchanged (ptToTest).

Example:

The following implementation example displays a message in the TwinCAT output window as soon as an invalid pointer is recognized. This implementation recognizes various types of invalid pointers. However, it cannot recognize all invalid pointers.

```
// 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

IF ptToTest=0 THEN
    ADSLOGSTR(ADSLOG_MSGTYPE_ERROR OR ADSLOG_MSGTYPE_STRING,'CheckPointer failed due to invalid destination address.','');
ELSIF iSize<=0 THEN
    ADSLOGSTR(ADSLOG_MSGTYPE_ERROR OR ADSLOG_MSGTYPE_STRING,'CheckPointer failed due to invalid size.','');
ELSIF iGran<=0 THEN
    ADSLOGSTR(ADSLOG_MSGTYPE_ERROR OR ADSLOG_MSGTYPE_STRING,'CheckPointer failed due to invalid granularity.','');
// -> Please note that the following memory area check is time consuming:
//ELSIF F_CheckMemoryArea(pData:=ptToTest,nSize:=DINT_TO_UDINT(iSize)) = E_TcMemoryArea.Unknown THEN
//    ADSLOGSTR(ADSLOG_MSGTYPE_ERROR OR ADSLOG_MSGTYPE_STRING,'CheckPointer failed due to unknown memory area.','');
END_IF
CheckPointer := ptToTest;
```

⚠ CAUTION

Consequence of an invalid pointer

The consequence of an invalid pointer is usually that the runtime is stopped as soon as any access takes place via this pointer. The CheckPointer function cannot usually prevent this. The purpose of this monitoring function is actually to make efficient cause diagnosis possible.

7.20 Object-oriented programming

TwinCAT 3 supports object-oriented programming (OOP) and provides the following functionalities and objects for this purpose:

- Function blocks ([Object Function block](#) [[▶ 171](#)])
- Methods ([Method object](#) [[▶ 173](#)])
- Properties ([Property object](#) [[▶ 179](#)])
- Interfaces ([Object Interface](#) [[▶ 185](#)], [Implementation of an interface](#) [[▶ 198](#)])
- Inheritance
 - Definition of function blocks as [Extensions of other function blocks](#) [[▶ 191](#)]
 - Definition of structures as [Extensions of other structures](#) [[▶ 197](#)]
 - Definition of interface as [Extensions of other interfaces](#) [[▶ 197](#)]

- [Method call \[► 199\]](#)
- [ABSTRACT concept \[► 201\]](#)

Basic idea of object-oriented programming

In object-oriented programming, the software is divided into objects. All descriptions relating to an object are combined in one element (a function block, for example). The descriptions include the data and the procedures associated with the object. In addition, an access interface to the object can be defined via methods and properties.

As a result, this programming approach develops objects that can be reused autonomously and independent of specific conditions. The elements can be used without modification in one or many applications.

Advantages of object-oriented programming

Object-oriented programming method offers many advantages.

By dividing the software into objects, a **clear**, well **structured** application can be developed. Thus, the application and the individual elements are easily **understandable** and easy to **expand**. The **reusability** of programming objects saves **time** and **costs** in the development and maintenance of applications.

General notes/recommendations

- When using object-oriented programming, care must be taken to find the correct "degree of object orientation" for the respective application.
 - With the help of OOP, the software is divided into objects, so that a clear, structured, reusable software can be developed.
 - However, if the division into the objects is excessively detailed, the comprehensibility of the program suffers.
 - Example: Using the inheritance feature, declarations and implementations can be passed on and thus reused by the subclass. In addition, division into a general and a specific class can support the understanding of the individual programming objects. This is the case with inheritance if the division into 'basic' and 'extended' is consistent. A real-world example is a basic motor and a more specific motor with additional functions. However, if a large number of inheritance levels are used and the function of the individual inheritance levels becomes unclear or imprecise, the application can become difficult to understand and maintain.
- If methods provide return values, these should be set in the method and evaluated at the point where the method is called.
- If methods, functions or properties return structured return types (e.g. a structure or a function block), these return types should be declared as "REFERENCE TO <structured type>".
 - Returning structured data as a direct return type ("<structured type>") is inefficient because the data is copied multiple times.
 - If "REFERENCE TO <structured type>" is used instead, on the one hand this is more efficient because the data is not copied, and on the other hand a single element of the structured data type can be accessed directly when the method/function/property is called.
 - For more information and an example see the [Object method \[► 173\]](#) description.
- Programming of object-oriented implementations should be event-based.
 - Program elements should not be called cyclically without reason.
 - Usually, it makes sense to call a program element when a certain event has occurred.
- A method or property call via an unassigned interface variable has the same effect as a NullPointer call. Before you can use the interface variable, you must assign a corresponding function block instance to it (instance of a function block that implements the interface).
 - To ensure that the interface variable does not correspond to a NullPointer, the interface variable can be checked for unequal 0 before it is used (e.g. `IF (iSample <> 0) THEN iSample.METH (); END_IF`).

OOP and UML

Does object-oriented programming (OOP) and UML always have to be used together?

- The combined use of OOP and UML offers many benefits (see next question), although it is not compulsory. Object-oriented programming of applications is also possible without using UML. Likewise, UML can be used in PLC projects, which are not based on object-oriented programming.

What are the benefits of using OOP and UML together?

- In order to make the most of OOP, the structure of an object-oriented software should be designed and created before the implementation (e.g. What classes are available, what is their relationship, what functionalities do they offer, etc.). Before, during and after programming, documentation helps to understand, analyze and maintain the software.
- As an analysis, design and documentation tool for software, UML offers options for planning, creating and documenting the application. UML is particularly suitable for object-oriented implementations, since modular software is particularly suitable for representation with the aid of a graphical language.
- For example, the class diagram is used for analyzing and creating the program structure. The more modular the software structure, the easier and more efficient the class diagram can be used (e.g. Graphical representation of separate function blocks with individual methods for providing the functionalities etc.).
- The state diagram can be used to specify the sequence of a system with discrete events. The more consistent the object- and event-orientation of the software structure, the more transparent and effective the state machines can be designed (e.g. The behavior of modules/systems is based on a state model with states (such as startup, production, pause); within the states corresponding functionalities are called, which are encapsulated in methods (such as startup, execute, pause) etc.).

See also:

- Documentation for: TF1910 TC3 UML
- Documentation samples: [Sample: Object-oriented program for controlling a sorting plant](#)

7.20.1 Object Function block

A function block is a [POU \[► 78\]](#), which returns one or several values when executed. The values of the output variables and the internal variables are retained until the next execution. This means that the function block may not return the same output values, if it is called repeatedly with the same input variables.

In the PLC project tree, function block POU's have the suffix (FB). The editor of a function block consists of the declaration part and the implementation part.

A function block is always called via an instance, which is a copy of the function block.

In addition to the functionality described in IEC 61131-3, in TwinCAT function blocks can also be used for the following object-oriented programming functionalities:

- Extending a function block ([Extending a function block \[► 191\]](#))
- Implementing interfaces ([Implementing an interface \[► 198\]](#))
- Methods ([Object Method \[► 173\]](#))
- Properties ([Object Property \[► 179\]](#))

The top line of the declaration part contains the following declaration:

```
FUNCTION_BLOCK <access specifier> <function block> | EXTENDS <function block> |
IMPLEMENTS <comma-separated list of interfaces>
```

● 8-byte alignment

i An 8-byte alignment was introduced with TwinCAT 3. Make sure that the alignment is appropriate if data are exchanged as an entire memory block with other controllers or software components (see [Alignment \[► 791\]](#)).

Automatic creation of interface elements in a function block

There are two ways of automatically generating elements of an interface that implements a function block in this function block.

1. If you specify an interface in the field **Implements** in the dialog **Add** when creating a new function block, TwinCAT automatically also adds the methods and properties of the interface to this function block.
2. If an existing function block implements an interface, you can use the command **Implement Interface** in order to generate the interface elements in the function block. The command **Implement Interface** can be found in the context menu of a function block in the project tree.

See also:

- TC3 User Interface documentation: [Command Implement interfaces \[► 1063\]](#)
- [Implementation of an interface \[► 198\]](#)

Calling a function block

The call always takes place via an instance of the function block. If a function block is called, only the values of the respective instance change.

Declaration of the instance:

```
<instance> : <function block>;
```

A variable of the function block is accessed as follows in the implementation part:

```
<instance>.<variable>
```

- From outside the function block instance, you can only access input and output variables of a function block, not the internal variables.
- Access to a function block instance is limited to the POU in which the instance is declared, unless you have declared the instance globally.
- You can assign the desired values to the function block variables when you call the instance.

Sample:

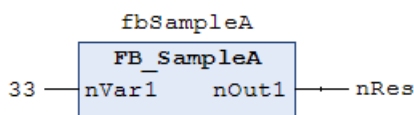
The function block FB_SampleA has the input variable nVar1 of type INT and the output variable nOut1. In the sample below, the variable nVar1 is called from the MAIN program.

ST:

```
PROGRAM MAIN
VAR
    fbSampleA : FB_SampleA;
END_VAR

fbSampleA.nVar1 := 33; (* FB_SampleA is called and the value 33 is assigned to the variable nVar1 *)
fbSampleA(); (* FB_SampleA is called, that's necessary for the following access to the output variable *)
nRes := fbSampleA.nOut1 (* the output variable nOut1 of the FB1 is read *)
```

FBD:



Assigning variable values during a call:

In the text-based languages IL and ST, you can assign values directly to input and/or output variables when the function block is called.

A value is assigned to an input variable with :=

A value is assigned to an output variable with =>

Sample:

The instance fbTimer of the timer function block is called with assignments for the input variables IN and PT. The output variable Q of the timer is then assigned to the variable bVarA

```
PROGRAM MAIN
VAR
    fbTimer : TOF;
    bIn     : BOOL;
    bVarA   : BOOL;
END_VAR
fbTimer(IN := bIn, PT := t#300ms);
bVarA := fbTimer.Q;
```




If you add a function block instance via the input assistant and the option **Insert with arguments** is enabled in the **Input Assistant** dialog, TwinCAT adds the call with all input and output variables. All you have to add is the desired value assignment. In the above example, TwinCAT adds the call as follows: CMD_TMR (IN:= , PT:= , Q=>).



Using the attribute **'is_connected'** on a local variable, you can determine at the time of the call in the function block instance whether a particular input receives an assignment from outside.

7.20.2 Object Method

Symbol: 

The object is used for object-oriented programming.

A method contains a sequence of statements. In contrast to a function, it is not an independent POU, but has to be assigned to a function block or a program.

You can use interfaces to organize methods.

Notes on methods

- All data of a method are temporary and are only valid while the method is executed (stack variables). This means that TwinCAT re-initializes all variables and function blocks, which you have declared in a method, with each call of the method.
- Like functions, methods can return a return value.
- According to the IEC 61131-3 standard, methods can have additional inputs and outputs, like normal functions. You assign the inputs and outputs when the method is called.
 - **From TwinCAT 3.1.4026:** Inputs without an explicitly specified initial value must be assigned when the method is called. Inputs with an explicitly specified initial value can be optionally assigned or ignored when the method is called.
- Access to the function block instance or program variables is permitted in the implementation part of a method.
- Use the THIS pointer to point to your own instance.
- It is not possible to access VAR_TEMP variables of the function block in a method.
- You can declare VAR_INST variables for which no reinitialization takes place when methods are called up again (see also [chapter Instance variables](#)).
- By using the return type "REFERENCE TO <structured type>", you can access a single element of the structured data type returned by the method directly when calling the method. For more information see section "[Access to a single element of a structured return type during method/function/property call \[► 177\]](#)".
- In principle, access to VAR_IN_OUT variables of a function block is possible in a method. Since this access is potentially risky, it should be used advisedly. Further information can be found in section "[Access to VAR_IN_OUT variables of a function block in a method/transition \[► 178\]](#)".

- Methods that are defined in an interface may only define input, output and VAR_IN_OUT variables, but they may not contain implementations.

Sample:

The code in the following sample causes TwinCAT to write the return value and the outputs of the method to locally declared variables.

Declaration part of "Method1" of the function block FB_Sample:

```
METHOD Method1 : BOOL
VAR_INPUT
  nIn1  : INT;
  bIn2  : BOOL;
END_VAR
VAR_OUTPUT
  fOut1 : REAL;
  sOut2 : STRING;
END_VAR
// <method implementation code>
```

MAIN program:

```
PROGRAM MAIN
VAR
  fbSample      : FB_Sample;
  bReturnValue  : BOOL;
  nLocalInput1 : INT;
  bLocalInput2 : BOOL;
  fLocalOutput1 : REAL;
  sLocalOutput2 : STRING;
END_VAR
bReturnValue := fbSample.Method1(nIn1 := nLocalInput1,
                                bIn2 := bLocalInput2,
                                fOut1 => fLocalOutput1,
                                sOut2 => sLocalOutput2);
```

Creating an object Method

1. Select a function block or a program in the **Solution Explorer** in the PLC project tree.
2. In the context menu select the command **Add > Method...**
 - ⇒ The dialog **Add Method** opens.
3. Enter a name and select a return type, the implementation language, and optionally an access modifier
4. Click on **Open**.
 - ⇒ The object is added to the PLC project tree and opens in the editor. The editor consists of the declaration editor at the top and the implementation part at the bottom.







Dialog Add method

Name	Name of the method The standard methods FB_Init and FB_Exit are offered in a selection list if they are not already inserted below the function block. If it is a derived function block, the selection list also offers all methods of the basic function block.
Return type	Type of the value that is returned
Implementation language	Implementation language selection list

Access modifier

Access modifier	<p>Regulates access to the data</p> <ul style="list-style-type: none"> • PUBLIC: Access is not restricted (equivalent to specifying no access modifier). • PRIVATE: Access to the method is restricted to the function block or the program respectively. • PROTECTED: Access to the method is restricted to the program or the function block and its derivatives respectively. • INTERNAL: Access to the method is limited to the namespace (the library). <p>In addition to these access modifiers, you can manually add the FINAL modifier to a method:</p> <ul style="list-style-type: none"> • FINAL: Overwriting the method in a derivative of the function block is not allowed. This means that the method may not be overwritten/extended in a possibly existing subclass.
Abstract	<p><input checked="" type="checkbox"/> : Indicates that the method has no implementation and that the implementation is provided by the derived FB.</p> <p>Background information on the ABSTRACT keyword can be found under ABSTRACT concept [► 201].</p>

Methods with a different access modifier than PUBLIC are marked with a signal symbol in the Solution Explorer in the PLC project tree.

Access modifier	Object icon	Signal symbol
PRIVATE		 (Lock)
PROTECTED		 (Star)
INTERNAL		 (Heart)



If you copy or move a method from a POU to an interface, TwinCAT automatically deletes the included implementations.

Special methods for a function block

FB_init	<p>Declarations are implicit by default. Explicit declaration are also possible. Contains initialization code for the function block, as defined in the declaration part of the function block.</p> <p>(Methods FB_init, FB_reinit and FB_exit [► 848])</p>
FB_reinit	<p>Explicit declaration required. This is called when the instance of the function block was copied (like during an online change). It re-initializes the new instance module.</p> <p>(Methods FB_init, FB_reinit and FB_exit [► 848])</p>
FB_exit	<p>Explicit declaration required.</p> <p>Call for each instance of the function block before another download or a reset or during an online change for all moved or deleted instances.</p> <p>(Methods FB_init, FB_reinit and FB_exit [► 848])</p>
Properties and interface properties	<p>They each consist of a Set and/or a Get accessor method.</p> <p>(Object Interface property [► 190], Object Property [► 179])</p>

Calling a method

Syntax:

```
<return value variable> := <POU name>.<method name>(<method input name> :=
<variable name> (, <further method input name> := <variable name> )* );
```

When you call the method, you assign transfer parameters to the input variables of the method. In doing so, observe the declaration. It is sufficient to specify the names of the input variables without considering their order in the declaration.

Sample:

The code in the following sample causes TwinCAT to write the return value and the outputs of the method to locally declared variables.

Declaration part of "Method1" of the function block FB_Sample:

```
METHOD Method1 : BOOL
VAR_INPUT
    nIn1 : INT;
    bIn2 : BOOL;
END_VAR
VAR_OUTPUT
    fOut1 : REAL;
    sOut2 : STRING;
END_VAR
// <method implementation code>
```

MAIN program:

```
PROGRAM MAIN
VAR
    fbSample : FB_Sample;
    bReturnValue : BOOL;
    nLocalInput1 : INT;
    bLocalInput2 : BOOL;
    fLocalOutput1 : REAL;
    sLocalOutput2 : STRING;
END_VAR
bReturnValue := fbSample.Method1(nIn1 := nLocalInput1,
                                bIn2 := bLocalInput2,
                                fOut1 => fLocalOutput1,
                                sOut2 => sLocalOutput2);
```

Access to a single element of a structured return type during method/function/property call

The following implementation can be used to directly access an individual element of the structured data type that is returned by the method/function/property when a method, function or property is called. A structured data type is, for example, a structure or a function block.

1. The return type of the method/function/property is defined as "REFERENCE TO <structured type>" (instead of just "<structured type>").
2. Note that with such a return type – if, for example, an FB-local instance of the structured data type is to be returned – the reference operator REF= must be used instead of the "normal" assignment operator :=.

The declarations and the sample in this section refer to the call of a property. However, they are equally transferrable to other calls that deliver return values (e.g. methods or functions).

Sample

Declaration of the structure ST_Sample (structured data type):

```
TYPE ST_Sample :
STRUCT
    bVar : BOOL;
    nVar : INT;
END_STRUCT
END_TYPE
```

Declaration of the function block FB_Sample:

```
FUNCTION_BLOCK FB_Sample
VAR
    stLocal      : ST_Sample;
END_VAR
```

Declaration of the property FB_Sample.MyProp with the return type "REFERENCE TO ST_Sample":

```
PROPERTY MyProp : REFERENCE TO ST_Sample
```

Implementation of the Get method of the property FB_Sample.MyProp:

```
MyProp REF= stLocal;
```

Implementation of the Set method of the property FB_Sample.MyProp:

```
stLocal := MyProp;
```

Calling the Get and Set methods in the main program MAIN:

```
PROGRAM MAIN
VAR
    fbSample      : FB_Sample;
    nSingleGet    : INT;
    stGet         : ST_Sample;
    bSet          : BOOL;
    stSet         : ST_Sample;
END_VAR

// Get - single member and complete structure possible
nSingleGet := fbSample.MyProp.nVar;
stGet      := fbSample.MyProp;

// Set - only complete structure possible
IF bSet THEN
    fbSample.MyProp REF= stSet;
    bSet              := FALSE;
END_IF
```

Through the declaration of the return type of the property MyProp as "REFERENCE TO ST_Sample" and through the use of the reference operator REF= in the Get method of this property, a single element of the returned structured data type can be accessed directly on calling the property.

```
VAR
    fbSample      : FB_Sample;
    nSingleGet    : INT;
END_VAR

nSingleGet := fbSample.MyProp.nVar;
```

If the return type were only to be declared as "ST_Sample", the structure returned by the property would first have to be assigned to a local structure instance. The individual structure elements could then be queried on the basis of the local structure instance.

```
VAR
    fbSample      : FB_Sample;
    stGet         : ST_Sample;
    nSingleGet    : INT;
END_VAR

stGet      := fbSample.MyProp;
nSingleGet := stGet.nVar;
```

Access to VAR_IN_OUT variables of the function block in a method/transition/property

In principle, the VAR_IN_OUT variables of a function block can be accessed in a method, transition or property of the function block. Note the following for this type of access:

- If the body or an action of the function block is called from outside the FB, the compiler ensures that the VAR_IN_OUT variables of the function block are assigned with this call.
- This is not the case if a method, transition or property of the function block is called, since the VAR_IN_OUT variables of the FB cannot be assigned within a method, transition or property call. Therefore, access to the VAR_IN_OUT variables might occur by calling the method/transition/property before the VAR_IN_OUT variables are assigned to a valid reference. Since this would mean invalid access at runtime, accessing the VAR_IN_OUT variables of the FB in a method, transition or property is potentially risky.

Therefore, the following warning with ID C0371 is issued if the VAR_IN_OUT variables of the FB are accessed in a method, transition or property:

„Warning: Access to VAR_IN_OUT <Var> declared in <POU> from external context <Method/Transition/Property>”

An adequate response to this warning could be to check the VAR_IN_OUT variables within the method/transition/property before it is accessed. The operator `__ISVALIDREF` can be used for this check, to ascertain whether a reference refers to a valid value. If this check is enabled, it can be assumed that the user is aware of the risk that potentially exists when the VAR_IN_OUT variables of the FB are accessed in a method/transition/property. Checking the reference is regarded as adequate handling of this risk. The corresponding warning can therefore be suppressed via attribute 'warning disable'.

A sample implementation of a method is shown below.

Function block FB_Sample:

```
FUNCTION_BLOCK FB_Sample
VAR_IN_OUT
    bInOut : BOOL;
END_Var
```

Methode FB_Sample.MyMethod:

```
METHOD MyMethod
VAR_INPUT
END_VAR

// The warning can be disabled here as the user is aware of the risk that the reference may not be
// valid by checking its validity
{warning disable C0371}

// Checking the VAR_IN_OUT reference, leave the current method in case of invalid reference
IF NOT __ISVALIDREF(bInOut) THEN
    RETURN;
END_IF


// Access to VAR_IN_OUT reference (only if the reference was confirmed as valid before)
bInOut := NOT bInOut;

// The warning may be restored at the end of the access area
{warning restore C0371}
```

See also:

- [Object-oriented programming \[► 169\]](#)
- [Object Interface \[► 185\]](#)
- [Implementation of an interface \[► 198\]](#)
- [Extending a function block \[► 191\]](#)
- [Method call \[► 199\]](#)
- [ABSTRACT concept \[► 201\]](#)
- [Reference Programming > Instance Variables - VAR_INST \[► 688\]](#)

7.20.3 Object Property

Symbol: 

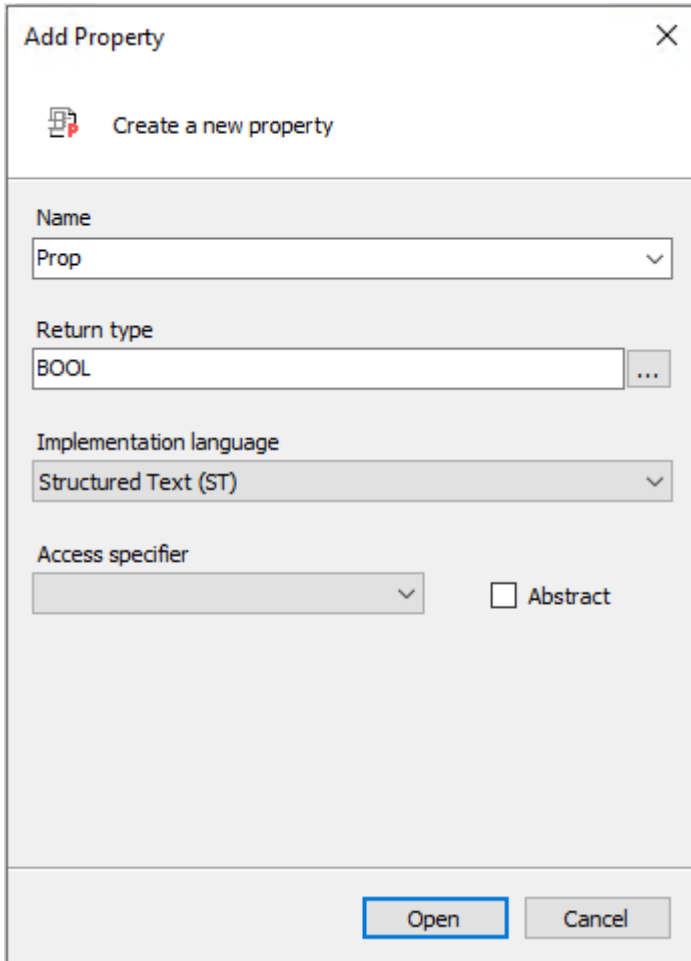
A property is an extension of the IEC 61131-3 standard and is a means for object-oriented programming. It consists of the accessor methods Get and Set. TwinCAT automatically calls these methods when a read or write access occurs to the function block that implements the property.

Object Creating a property

1. Select a function block or a program in the **Solution Explorer** in the PLC project tree.
2. In the context menu select the command **Add > Property...**

- ⇒ The **Add Property** dialog opens.
- 3. Enter a name and select a return type, the implementation language, and optionally an access modifier.
- 4. Click on **Open**.
- ⇒ The object is added to the PLC project tree and opens in the editor.

Dialog Add property









Name	Name of the property
Return type	Type of the value that will be returned (default type or structured type)
Implementation language	Implementation language selection list

Access modifier

Access specifier	<p>Regulates access to the data</p> <ul style="list-style-type: none"> • PUBLIC: Access is not restricted (equivalent to specifying no access modifier). • PRIVATE: Access to the property is restricted to the function block or the program respectively. • PROTECTED: Access to the property is restricted to the program or the function block and its derivatives respectively. • INTERNAL: Access to the property is limited to the namespace (the library). <p>In addition to these access modifiers, you can manually add the FINAL modifier to a property:</p> <p>FINAL: Overwriting the property in a derivative of the function block is not allowed. This means that the property may not be overwritten/extended in a possibly existing subclass.</p>
Abstract	<p><input checked="" type="checkbox"/> : Indicates that the property has no implementation and that the implementation is provided by the derived FB.</p> <p>Background information on the ABSTRACT keyword can be found under ABSTRACT concept [► 201].</p>

Properties with a different access modifier than PUBLIC are marked with a signal symbol in the Solution Explorer in the PLC project tree.

Access modifier	Object icon	Signal symbol
PRIVATE		 (Lock)
PROTECTED		 (Star)
INTERNAL		 (Heart)

In addition, a property can contain local variables. However, a property cannot contain additional inputs. In contrast to a function or method, it also cannot contain additional outputs.



If you copy or move a property from a POU to an interface, TwinCAT automatically deletes the included implementations.

Get and Set accessors

TwinCAT automatically adds the Get and Set accessor methods below the property object in the PLC project tree. The **Add** command can be used to add them explicitly.

TwinCAT calls the Set accessor when write access to the property occurs, i.e. when you use the property name as input parameter.

TwinCAT calls the Get accessor when read access to the property occurs, i.e. when you use the property name as output parameter.

Please note that you have to implement the accessor methods in order to access the property.

Implementation sample

Declaration of the function block FB_Sample

```
FUNCTION_BLOCK FB_Sample
VAR
  nVar : INT;
END_VAR
```

```
nVar := nVar + 1;
```

Declaration of the property nValue

```
PROPERTY PUBLIC nValue : INT
```

Implementation of the accessor method FB_Sample.nValue.Set

```
nVar := nValue;
```

Implementation of the accessor method FB_Sample.nValue.Get

```
nValue := nVar;
```

Calling the property nValue

```
PROGRAM MAIN
VAR
    fbSample : FB_Sample;
END_VAR

fbSample();
If fbSample.nValue > 500 THEN
    fbSample.nValue := 0;
END_IF;
```

If you only want to use the property for read access or only for write access, you can delete the unused accessor.



You can add access specifiers to the accessor methods in the following places:

- Through manual entries in the declaration part of the accessor.
- In the **Add 'get' accessor** or **Add 'set' accessor** dialog, when you add the accessor explicitly with the **Add** command.



Compatibility warning for properties of type REFERENCE

In TC3.1 Build 4022 the calling behavior of properties defined with the return type 'REFERENCE TO <...>' changes.

For write accesses using ':=' , with versions < 3.1.4022.0 the set accessor is called so that the reference is written. With versions >= 3.1.4022.0, however, the get-accessor is called so that the value is written. To assign the reference with versions >= 3.1.4022.0, the reference assignment operator 'REF= [[▶ 629](#)]' must be used.

Monitoring for properties in online mode

Pragmas are available for monitoring for properties in online mode, which should be added at the top of the definition property ([Attribute 'monitoring'](#) [[▶ 810](#)]):

- {attribute 'monitoring':='variable'}: With each access to the property, TwinCAT stores the actual value in a variable and displays the value of this variable. This value may become obsolete, if the code no longer accesses the property.
- {attribute 'monitoring' := 'call'}: Each time the value is displayed, TwinCAT calls the code of the Get accessor. If this code contains a side effect, the side effect is executed by the monitoring.

You can monitor a property using the following features:

- Inline-Monitoring
Requirement: In the TwinCAT options in the category **TwinCAT > PLC Programming Environment > Text Editor** the option **Enable Inline-Monitoring** is activated in the tab **Monitoring** .
- Watch List ([Using Watchlists](#) [[▶ 226](#)])

Access to a single element of a structured return type during method/function/property call

The following implementation can be used to directly access an individual element of the structured data type that is returned by the method/function/property when a method, function or property is called. A structured data type is, for example, a structure or a function block.

1. The return type of the method/function/property is defined as "REFERENCE TO <structured type>" (instead of just "<structured type>").

- Note that with such a return type – if, for example, an FB-local instance of the structured data type is to be returned – the reference operator REF= must be used instead of the "normal" assignment operator :=.

The declarations and the sample in this section refer to the call of a property. However, they are equally transferrable to other calls that deliver return values (e.g. methods or functions).

Sample

Declaration of the structure ST_Sample (structured data type):

```
TYPE ST_Sample :
STRUCT
  bVar : BOOL;
  nVar : INT;
END_STRUCT
END_TYPE
```

Declaration of the function block FB_Sample:

```
FUNCTION_BLOCK FB_Sample
VAR
  stLocal : ST_Sample;
END_VAR
```

Declaration of the property FB_Sample.MyProp with the return type "REFERENCE TO ST_Sample":

```
PROPERTY MyProp : REFERENCE TO ST_Sample
```

Implementation of the Get method of the property FB_Sample.MyProp:

```
MyProp REF= stLocal;
```

Implementation of the Set method of the property FB_Sample.MyProp:

```
stLocal := MyProp;
```

Calling the Get and Set methods in the main program MAIN:

```
PROGRAM MAIN
VAR
  fbSample : FB_Sample;
  nSingleGet : INT;
  stGet : ST_Sample;
  bSet : BOOL;
  stSet : ST_Sample;
END_VAR

// Get - single member and complete structure possible
nSingleGet := fbSample.MyProp.nVar;
stGet := fbSample.MyProp;

// Set - only complete structure possible
IF bSet THEN
  fbSample.MyProp REF= stSet;
  bSet := FALSE;
END_IF
```

Through the declaration of the return type of the property MyProp as "REFERENCE TO ST_Sample" and through the use of the reference operator REF= in the Get method of this property, a single element of the returned structured data type can be accessed directly on calling the property.

```
VAR
  fbSample : FB_Sample;
  nSingleGet : INT;
END_VAR

nSingleGet := fbSample.MyProp.nVar;
```

If the return type were only to be declared as "ST_Sample", the structure returned by the property would first have to be assigned to a local structure instance. The individual structure elements could then be queried on the basis of the local structure instance.

```
VAR
  fbSample : FB_Sample;
  stGet : ST_Sample;
  nSingleGet : INT;
END_VAR
```

```
stGet      := fbSample.MyProp;
nSingleGet := stGet.nVar;
```

Access to VAR_IN_OUT variables of the function block in a method/transition/property

In principle, the VAR_IN_OUT variables of a function block can be accessed in a method, transition or property of the function block. Note the following for this type of access:

- If the body or an action of the function block is called from outside the FB, the compiler ensures that the VAR_IN_OUT variables of the function block are assigned with this call.
- This is not the case if a method, transition or property of the function block is called, since the VAR_IN_OUT variables of the FB cannot be assigned within a method, transition or property call. Therefore, access to the VAR_IN_OUT variables might occur by calling the method/transition/property before the VAR_IN_OUT variables are assigned to a valid reference. Since this would mean invalid access at runtime, accessing the VAR_IN_OUT variables of the FB in a method, transition or property is potentially risky.

Therefore, the following warning with ID C0371 is issued if the VAR_IN_OUT variables of the FB are accessed in a method, transition or property:

„Warning: Access to VAR_IN_OUT <Var> declared in <POU> from external context <Method/Transition/Property>”

An adequate response to this warning could be to check the VAR_IN_OUT variables within the method/transition/property before it is accessed. The operator `__ISVALIDREF` can be used for this check, to ascertain whether a reference refers to a valid value. If this check is enabled, it can be assumed that the user is aware of the risk that potentially exists when the VAR_IN_OUT variables of the FB are accessed in a method/transition/property. Checking the reference is regarded as adequate handling of this risk. The corresponding warning can therefore be suppressed via attribute 'warning disable'.

A sample implementation of a method is shown below.

Function block FB_Sample:

```
FUNCTION_BLOCK FB_Sample
VAR_IN_OUT
    bInOut : BOOL;
END_Var
```

Methode FB_Sample.MyMethod:

```
METHOD MyMethod
VAR_INPUT
END_VAR

// The warning can be disabled here as the user is aware of the risk that the reference may not be
// valid by checking its validity
{warning disable C0371}

// Checking the VAR_IN_OUT reference, leave the current method in case of invalid reference
IF NOT __ISVALIDREF(bInOut) THEN
    RETURN;
END_IF

// Access to VAR_IN_OUT reference (only if the reference was confirmed as valid before)
bInOut := NOT bInOut;

// The warning may be restored at the end of the access area
{warning restore C0371}
```

Initialization example:

In the following sample, an input variable and a property are initialized by a function block that has an [FB_init method](#) [► 850] with an additional parameter.

Function block FB_Sample:

```
FUNCTION_BLOCK FB_Sample
VAR_INPUT
    nInput      : INT;
END_VAR
VAR
```

```

    nLocalInitParam : INT;
    nLocalProp      : INT;
END_VAR

```

Method FB_Sample.FB_init:

```

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)
    nInitParam   : INT;
END_VAR
nLocalInitParam := nInitParam;

```

Property FB_Sample.nMyProperty and the associated Set function:

```

PROPERTY nMyProperty : INT
nLocalProp := nMyProperty;

```

Program MAIN:

```

PROGRAM MAIN
VAR
    fbSample : FB_Sample(nInitParam := 1) := (nInput := 2, nMyProperty := 3);
    aSample  : ARRAY[1..2] OF FB_Sample[(nInitParam := 4), (nInitParam := 7)]
              := [(nInput := 5, nMyProperty := 6), (nInput := 8, nMyProperty := 9)];
END_VAR

```

Initialization result:

- fbSample
 - nInput = 2
 - nLocalInitParam = 1
 - nLocalProp = 3
- aSample[1]
 - nInput = 5
 - nLocalInitParam = 4
 - nLocalProp = 6
- aSample[2]
 - nInput = 8
 - nLocalInitParam = 7
 - nLocalProp = 9

See also:

- [Object Interface property \[► 190\]](#)
- [Object-oriented programming \[► 169\]](#)
- [Extending a function block \[► 191\]](#)
- [Reference programming > Methods FB_init, FB_reinit and FB_exit \[► 848\]](#)

7.20.4 Object Interface

Symbol: 

Keyword: INTERFACE

An interface is a tool for object-oriented programming. The object **Interface** describes a set of method and property prototypes. Prototype in this context means that the methods and properties only contain declarations, but no implementation.

In this way you can use various function blocks with common properties in the same way.

You can add the objects **Interface property** and **Interface method** to the **Interface** object.


Creating an object Interface

1. Select a folder in the **Solution Explorer** in the PLC project tree.
2. In the context menu select the command **Add > Interface...**
 - ⇒ The **Add Interface** dialog opens.
3. Enter a name, and optionally select an interface to be extended.
4. Click on **Open**.
 - ⇒ The interface is added to the PLC project tree and opens in the editor.

Dialog Add Interface

Name	Interface name
------	----------------

Inheritance

Advanced	<input checked="" type="checkbox"/> : Extends the interface that is entered in the input field or selected via the input assistant  . This means that all interface methods, which the new interface extends, are also available in the new interface.
----------	---

Interface applications

1. Checking the interface declaration via the compiler

- In an interface (e.g. I_Sample) you declare the methods and properties (including return type, inputs etc.), which are to be associated with this interface.
- In the function blocks, which are to correspond to this interface and should therefore make the corresponding methods and properties available, you implement the interface I_Sample.
 - The function block contains the interface in the IMPLEMENTS list within its declaration part (e.g. FB_Sample IMPLEMENTS I_Sample).
 - A function block can implement one or several interfaces (e.g. FB_Sample IMPLEMENTS I_Sample1, I_Sample2).
- A function block, which implements an interface, must contain all methods and properties that are defined in this interface (interface methods and properties). The declaration of the methods and properties must match the declaration in the interface exactly (name, return type, inputs, outputs).
- The function blocks add function block-specific code to the interface methods and interface properties. If an interface is implemented by several function blocks, you can use the same method with the parameters but different implementation code in different function blocks.
- For function blocks, which implement one or several interfaces, the compiler checks whether the function blocks meet the respective interface declarations. If the element declarations in the interface and in the function block differ, or if the interface contains further elements, which are not included in the function block, the compiler reports an error.

2. Calling methods and properties of a function block instance via an interface variable

In addition to automatic verification of the interface declaration via the compiler, you can use interfaces to call an interface method or interface property of a function block instance via an interface variable.

- First, instantiate the interface (e.g. `iSample : I_Sample;`) and the function block(s), which implement(s) the interface correctly (see use case 1: Checking the interface declaration via the compiler).
- You can then assign the interface variable to each instance of a function block, which implements the interface correctly. If an interface variable has not yet been assigned, the variable contains the value 0 in online mode.
- In the last step, you can call an interface method or property via the interface variable. The method or property is called for the function block, to which the interface refers.

- Such an implementation enables the use of different, but similar function blocks via the interface variable in a consistent manner. Depending on the project state you can assign a particular function block instance to the interface variable, for example, so that the call of the interface methods and properties is identical, although a different function block instance is used, depending on the project state.



An interface type variable has to be assigned the instance of a function block, before a method or property can be called via the interface variable.

Notes

- You cannot declare variables within an interface. An interface has no implementation part and no actions. Only a collection of methods and properties is defined, which contain declaration code, but no implementation code.
- An interface type variable is a reference to instances of function blocks. TwinCAT always treats variables, which are defined as an interface type, as references.

Interface references and Online Change

- Interface references are automatically redirected from TwinCAT 3.0 build 3100 , so that the correct interface is always referenced, even in the event of an online change. This requires additional code and time, which may cause issues, depending on the number of affected objects. Before the online change is performed, programmers are therefore shown the number of affected variables and interface references, so that they can decide whether to go ahead with the online change or abort it.

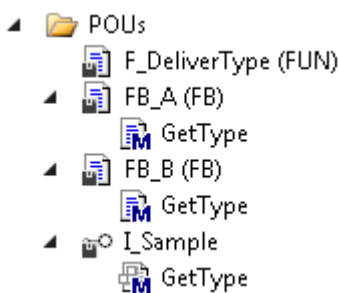
Checking interface variables

- The operator `__ISVALIDREF` can only be used for operands of type `REFERENCE TO`. This operator cannot be used for checking interface variables. To check whether an interface variable was already assigned a function block instance, you can check the interface variable for not equal to 0 (`IF iSample <> 0 THEN ...`).

Extended monitoring of an interface variable

- Advanced options for monitoring/debugging an interface variable are available from TwinCAT 3.1 Build 4024. An interface variable in the monitoring area (declaration editor, watch list) can be expanded. The symbol path and online data of the currently assigned FB instance are then displayed below the interface variable.

Example 1



Interface declaration:

- You have added the interface `I_Sample` to your project. Add the method `GetType` with the return type `STRING` to the interface.
- `I_Sample` and `GetType` contain no implementation code. The method `GetType` contains only the required (variable) declarations (e.g. return type). You can program out the method `GetType` later on in the function block that implements the interface `I_Sample`.

```
INTERFACE I_Sample
```

Method `I_Sample.GetType`:

```
METHOD GetType : STRING
```

Interface implementation:

- If you subsequently add a function block to the project and enter the interface I_Sample in the field **Implements** in the dialog **Add**, TwinCAT also automatically adds the method GetType to this function block. Here you can implement function block-specific code in the methods.
- The function blocks FB_A and FB_B both implement the interface I_Sample:

```
FUNCTION_BLOCK FB_A IMPLEMENTS I_Sample
```

```
FUNCTION_BLOCK FB_B IMPLEMENTS I_Sample
```

- Both function blocks therefore have to include a method with the name GetType and the return type STRING. Otherwise the compiler reports an error (see use case 1 in section [Interface application](#) [► 186]s).

Method FB_A.GetType:

```
METHOD GetType : STRING
```

```
GetType := 'FB_A';
```

Method FB_B.GetType:

```
METHOD GetType : STRING
```

```
GetType := 'FB_B';
```

Interface application:

- A function F_DeliverType contains the declaration of an input variable of the type of interface I_Sample. Within the function, the interface method GetType is called via the interface variable iSample. In this case, whether FB_A.GetType or FB_B.GetType is called depends on the transferred function block type (see application 2 in section [Interface application](#) [► 186]s).

```
FUNCTION F_DeliverType : STRING
VAR_INPUT
    iSample : I_Sample;
END_VAR

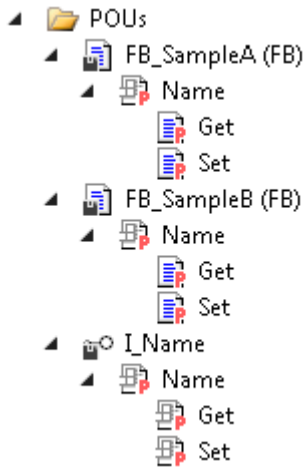
F_DeliverType := iSample.GetType();
```

- Instances of function blocks, which implement the interface I_Sample (e.g. FB_A and FB_B), can be assigned to the input variable of the function F_DeliverType.
- Examples of function calls:
 - If the function block instance fbA is transferred to the function F_DeliverType, the method fbA.GetType will be called inside the function since the interface variable iSample points to the function block instance fbA. This method call delivers the return value 'FB_A', which in turn is returned by the function F_DeliverType and assigned in the main program to the variable sResultA.
 - Accordingly, sResultB receives the value 'FB_B', since the method fbB.GetType is called inside the function F_DeliverType.

```
PROGRAM MAIN
VAR
    fbA      : FB_A;
    fbB      : FB_B;
    sResultA : STRING;
    sResultB : STRING;
END_VAR

sResultA := F_DeliverType(iSample := fbA); // call with instance of type FB_A
sResultB := F_DeliverType(iSample := fbB); // call with instance of type FB_B
```


Example 2



Interface declaration:

- You have added the interface I_Name to your project. Add the property Name with the return type STRING to the interface. The property has the accessor methods Get and Set. The Get accessor can be used to read the name of any object from a function block that implements the interface. The Set accessor is used to write the name into this function block.
- I_Name and Name contain no implementation code. The property Name only contains the required declaration (return type). Therefore you cannot process the Get and Set methods inside the interface definition, but you can do this later in the function block that implements the interface I_Name.

```
INTERFACE I_Name
```

Property I_Name.Name:

```
PROPERTY Name : STRING
```

Interface implementation:

- The function blocks FB_SampleA and FB_SampleB implement the interface I_Name.
- If the interface is specified, for example, when creating the function blocks in the dialog **Add**, TwinCAT automatically adds the property Name with the Get and Set methods under the function blocks FB_SampleA and FB_SampleB.
- You can edit the accessor methods underneath the function blocks, for example so that the variable sVar1 is read and you thus obtain the name of an object. In FB_SampleB, which implements the same interface I_Name, you can implement the Get method code, which then returns the name of another object. The Set method can be used to write the name, which the MAIN program supplies ('abc'), into the function block FB_SampleB.
- The function blocks FB_SampleA and FB_SampleB each implement the interface I_Name:

```
FUNCTION_BLOCK FB_SampleA IMPLEMENTS I_Name
VAR
  sVar1 : STRING := 'My name is A.';
END_VAR

FUNCTION_BLOCK FB_SampleB IMPLEMENTS I_Name
VAR
  sVar2 : STRING := 'My name is B.';
END_VAR
```

- Both function blocks must therefore contain a property with the name Name and the return type STRING. The property must have a Get and a Set method. Otherwise the compiler reports an error (see use case 1 in section [Interface applications \[▶ 186\]](#)).

FB_SampleA.Name.Get:

```
Name := sVar1;
```

FB_SampleA.Name.Set:

```
sVar1 := Name;
```

FB_SampleB.Name.Get:

```
Name := sVar2;
```

FB_SampleB.Name.Set:

```
sVar2 := Name;
```

Interface application:

- The properties of the function blocks can be accessed both via the corresponding function block instances and via an interface variable of the type I_Name. The prerequisite for access via an interface variable is that this variable has been assigned a specific function block instance beforehand that implements the interface I_Name.

```
PROGRAM MAIN
VAR
    iName      : I_Name;

    fbSampleA : FB_SampleA;
    sNameA    : STRING;      // will be 'My name is A.'

    fbSampleB : FB_SampleB;
    sNameB    : STRING;      // will be 'My name is B.' after first cycle
                                // and will be 'New name' afterwards
END_VAR

// assign FB instance fbSample1 to interface variable
iName := fbSampleA;


// access to name property of fbSample1 via interface variable (Get)
sNameA := iName.Name;

// access to name property of fbSample2 via FB instance (Get and Set)
sNameB := fbSampleB.Name;
fbSampleB.Name := 'New name';
```

See also:

- [Implementing an interface \[► 198\]](#)
- [Extending an interface \[► 197\]](#)

7.20.4.1 Object Interface method

Symbol: 

An interface method is a means of object-oriented programming. The **Interface method** object can be added to an interface via the command **Add > Method...**


If a method is added under an interface, you can only add and instantiate variable declarations (input, output and input/output variables) in this method.

Program code can only be added to the method, once a function block "implements" the interface that belongs to the method. TwinCAT then adds the method under the function block.

See also:

- [Object Interface \[► 185\]](#)
- [Object Method \[► 173\]](#)
- [Implementation of an interface \[► 198\]](#)

7.20.4.2 Object Interface property

Symbol: 

An interface property is a tool for object-oriented programming. The object **Interface property** is added to an interface via the command **Add > Property...** in order to extend the description of the interface with the accessor method Get and/or Set. No implementation code is included for the accessor methods in the interface property. If you delete the Set accessor, only read access is available for the property, not write access.

The Get accessor is used for read access to the property.
The Set accessor is used for write access to the property.

If the property has no Get and/or Set, this accessor can be added to the interface property with the command **Add**.

i If you extend a function block or a program with an interface that contains properties, TwinCAT automatically adds this and the corresponding Get and/or Set accessors in the PLC project tree under the POU. You can then implement the code in the Get and/or Set accessors.

See also:

- [Object Interface \[► 185\]](#)
- [Object Property \[► 179\]](#)
- [Implementation of an interface \[► 198\]](#)

7.20.5 Extending a function block

The extension of a function block is based on the concept of inheritance in object-oriented programming. A derived function block "extends" a basic function block for this purpose and thus basically obtains ("inherits") the properties and functionalities of the basic function block – in addition to its own properties and functionalities.

In TwinCAT, the term "function block" can be used interchangeably with "class". A derived function block can therefore be referred to as "subclass", and the basic function block as "base class".

Observe the following information:

- [Inheritance principle \[► 193\]](#)
- [Use cases for inherited elements \[► 194\]](#)

i Number of extensions for each basic function block

The number of extensions for each basic function block is unlimited. A function block can therefore be extended and customized with several other function blocks.

- Possible:

```
FUNCTION_BLOCK FB_Sub1 EXTENDS FB_Base
FUNCTION_BLOCK FB_Sub2 EXTENDS FB_Base
FUNCTION_BLOCK FB_Sub3 EXTENDS FB_Base
```
-

i Number of inheritance levels

The number of inheritance levels is unlimited. A function block that extends another function block can therefore be customized with a further function block, etc.

- Possible:

```
FUNCTION_BLOCK FB_Sub EXTENDS FB_Base
FUNCTION_BLOCK FB_SubSub EXTENDS FB_Sub
FUNCTION_BLOCK FB_SubSubSub EXTENDS FB_SubSub
```
-

● Multiple inheritance is not allowed

i Multiple inheritance is not allowed for function blocks. A function block cannot extend more than one other function block.


Exception: A function block can implement several interfaces and an interface can extend several other interfaces.

- Not possible:
FUNCTION_BLOCK FB_Sub EXTENDS FB_Base1, FB_Base2
- Possible:
FUNCTION_BLOCK FB_Sample IMPLEMENTS I_Sample1, I_Sample2
- Possible:
INTERFACE I_Sub EXTENDS I_Base_1, I_Base_2

● Overloaded

i Overloading of methods is not possible. Therefore, if you overwrite or extend a base class method (same method name) in a subclass, the method declaration must match the base class declaration (access modifier, return type, variable interface).

Extending a basic function block with a new function block

- ✓ The currently open project has a basic function block, for example FB_Sample, which is to be extended with a new function block.
- 1. Select the PLC project object or a subfolder in the PLC project tree, and in the context select the command menu **Add > POU...**
 - ⇒ The **Add POU** dialog opens.
- 2. Enter a name for the new function block in the Name input field, for example FB_SampleEx.
- 3. Select **Function Block**.
- 4. Select **Extends** and click on .
- 5. In the input assistant, from the **Functionblocks** category under the project select the POU(FB) to be used as basic function block, for example "FB_Sample", and click **OK**.
- 6. Optionally you can select an **Access specifier** for the new function block from the combo box.
- 7. From the **Implementation language** select "Structured Text (ST)", for example.
- 8. Click on **Open**.
 - ⇒ TwinCAT adds the function block FB_SampleEx in the PLC project tree, and the editor opens. The first line says:
FUNCTION_BLOCK FB_SampleEx EXTENDS FB_Sample
The function block FB_SampleEx extends the basic function block FB_Sample.

Extending a basic function block with an existing function block

- ✓ The currently open project has a basic function block, for example "FB_Sample", and a further function block, for example "FB_SampleEx", which is not yet derived from the basic function block. The function block FB_SampleEx is to extend the basic function block, i.e.: FB_SampleEx is to extend FB_Sample.
- 1. Double-click on the function block FB_SampleEx in the PLC project tree.
 - ⇒ The POU editor opens.
- 2. Extend the existing entry in the top row, FUNCTION_BLOCK FB_SampleEx, with EXTENDS FB_Sample.
 - ⇒ The function block FB_SampleEx extends the basic function block FB_Sample.

See also:

- [Object Function block \[► 171\]](#)
- [Object Property \[► 179\]](#)
- [Object Method \[► 173\]](#)
- [Object Action \[► 87\]](#)

- [Object Transition \[▶ 88\]](#)
- Reference Programming: [SUPER \[▶ 692\]](#)
- Reference Programming: [THIS \[▶ 694\]](#)

7.20.5.1 Inheritance principle

Content of the inheritance

A derived function block inherits all data, methods, properties, actions and transitions, which are defined in the basic function block. Note the access options to inherited elements, which are defined via access modifiers.

Access options to inherited elements

To what extent a subclass can access inherited methods or properties within its scope depends on the access modifier, with which the method or property is defined in the base class.

Methods and properties, which are defined in the base class with the access modifier PRIVATE, cannot be called within the scope of the subclass, nor can they be overwritten or extended by the subclass. Private methods and properties are available for the subclass only in so far as that they are executed for the instance of the subclass, if they are called within the implementation of the base class.

Example:

The base class has a PUBLIC and a PRIVATE method. The PUBLIC method calls the PRIVATE method in its implementation. The PUBLIC method can be called by the subclass called, so that the PRIVATE method is called implicitly at the same time. However, the PRIVATE method cannot be actively called, overwritten or extended by the subclass.

The following access modifiers available for specifying access options to a method or property:

PUBLIC	Corresponds to the specification of no access modifier. The element (method or property) can be called from outside the function block. Therefore, the element can also be accessed by a subclass.
PRIVATE	Access to the element is limited to the function block. Access from outside the function block is not possible. This means that a subclass cannot access the element either. The subclass can therefore neither call the element, nor overwrite or extend it.
PROTECTED	Access to the element is limited to the function block and its derivatives. A subclass can access the element and can therefore call, extend or overwrite it. Access from outside this "inheritance family" is not possible.
INTERNAL	Access to the element is limited to the namespace (the library). Access from outside the namespace is not possible. Therefore, the element cannot be overwritten or extended from outside the namespace.

Extending or overwriting of inherited elements

- A derived function block can extend or overwrite the methods, properties, actions and transitions defined in the basic function block, if a corresponding access modifier for the elements is used in the base class.
- In order to be able to extend or overwrite an element, the element must be declared in the subclass in the same way as in the base class:
 - same name
 - same access modifier (for methods and properties)
 - same variable interface (e.g. method inputs/outputs)
 - same return type (for methods and properties)
- When an element is extended or overwritten, in the subclass only the implementation part is adjusted, in order to adjust the behavior of the element.

Engineering tip: The following engineering support is provided for extending or overwriting methods, properties, actions and transitions inherited by the function block: When you add a method, property etc. to the derived function block, the **Name** of the Add dialog (e.g. **Add Method**, **Add Property**) offers a drop-down list with a choice of methods, properties etc. used in the basic function block. If, for example, you select one of these methods in the **Add Method** dialog, the other declaration settings for the method (return type, access modifier) are automatically applied to the method declaration of the base class. When you confirm the dialog, the method is created according to these declarations. You can customize the implementation part of the method such that it matches the desired behavior of the subclass.

Further information on extending and overwriting can be found in section "[Use cases for inherited elements \[► 194\]](#)".

Further considerations relating to extensions

- An instance of the derived function block can be used in any context in which TwinCAT expects a function block of the type of the basic function block.
- A derived function block must not contain any function block variables with the same names as declared by the basic function block. The compiler reports this as an error.
The only exception: If you have declared a variable as VAR_TEMP in the basic function block, the derived function block may define a variable with the same name. The derived function block can then no longer access the variable of the basic function block.
- Within the scope of the derived function block, the variables and elements (e.g. methods, properties, actions, transitions) of the basic function block can be addressed directly by using the SUPER pointer.

7.20.5.2 Use cases for inherited elements

In general, inherited elements to which the subclass has appropriate access (see section "[Access options to inherited elements \[► 193\]](#)") can be used in three different ways:

- Inherited elements can be used unchanged.
- Inherited elements can be overwritten.
- Inherited elements can be extended.

These three use cases are explained below using the example of the "Method" element.

Unchanged use

- Requirement: The subclass requires exactly the same implementations that have already been programmed in the basic class method.
- Implementation: In this case, the method is **not** created for the subclass.
- Consequence: The subclass uses the method implementation of the base class.
- Example:
 - The base class must control an axis for executing a process.
 - The same requirement applies to the subclass: the subclass must also control the axis.
 - In this case, the method ExecuteProcess is **not** created for the subclass. If the method is called for an instance of the subclass (fbSub.ExecuteProcess (...)), the basic implementation of the method is called automatically (FB_Base.ExecuteProcess). As a result, the subclass benefits from the implementations that have already been implemented in the base class.

Function block FB_Base:

```
FUNCTION_BLOCK FB_Base
VAR
    fbAxis    : FB_Axis;
END_VAR
```

Method FB_Base.ExecuteProcess:

```
METHOD ExecuteProcess : BOOL
VAR_INPUT
    bExecuteProcess : BOOL;
END_VAR
```

```
// Calling axis module by passing input parameter "bExecuteProcess" of this method to the input
parameter "bExecute" of method "Execute"
fbAxis.Execute(bExecute := bExecuteProcess);

// Setting the return value of this method as inverted error signal of the axis module
ExecuteProcess := NOT fbAxis.Error;
```

Function block FB_Sub:

```
FUNCTION_BLOCK FB_Sub EXTENDS FB_Base
VAR
END_VAR
```

Method FB_Sub.ExecuteProcess:

[does not exist]

Overwrite

- **Requirement:** Compared with the base class, the subclass requires completely different instructions in the method.
- **Implementation:** In this case, the method for the subclass is created and filled with other instructions in the implementation part. Compared with the base class method, only the implementation part differs – the declaration part must be identical.
- **Consequence:** The subclass uses its own implementation of the method. The subclass has overwritten the base class method.
- **Example:**
 - The base class must control an axis for executing a process.
 - In contrast, the subclass does not have to control an axis but a cylinder during the process execution.
 - In this case, the method ExecuteProcess is created for the subclass. The implementation part of the method is programmed with the required instructions, which have a completely different effect compared to the basic implementation.

Function block FB_Base:

```
FUNCTION_BLOCK FB_Base
VAR
    fbAxis : FB_Axis;
END_VAR
```

Method FB_Base.ExecuteProcess:

```
METHOD ExecuteProcess : BOOL
VAR_INPUT
    bExecuteProcess : BOOL;
END_VAR

// Calling axis module by passing input parameter "bExecuteProcess" of this method to the input
parameter "bExecute" of method "Execute"
fbAxis.Execute(bExecute := bExecuteProcess);

// Setting the return value of this method as inverted error signal of the axis module
ExecuteProcess := NOT fbAxis.Error;
```

Function block FB_Sub:

```
FUNCTION_BLOCK FB_Sub EXTENDS FB_Base
VAR
    fbCylinder : FB_Cylinder;
END_VAR
```

Method FB_Sub.ExecuteProcess:

```
METHOD ExecuteProcess : BOOL
VAR_INPUT
    bExecuteProcess : BOOL;
END_VAR

// Calling cylinder module by passing input parameter "bExecuteProcess" of this method to the input
parameter "bExecute" of method "Execute"
fbCylinder.Execute(bExecute := bExecuteProcess);
```

```
// Setting the return value of this method as inverted error signal of the cylinder module
ExecuteProcess := NOT fbCylinder.Error;
```

Extension

- **Requirement:** The subclass requires both the implementation as already implemented in the base class, as well as additional instructions that are specific to the subclass.
- **Implementation:** In this case, the method for the subclass is created and filled with the additionally required instructions in the implementation part. At the desired position within the subclass method, the base class method is called via `SUPER^.SampleMethod (...)`. This call executes the original base class method. The additional instructions within the subclass method also execute additional instructions that are specifically required for the subclass.
- **Consequence:** The subclass uses its own additional implementation as well as the implementation of the base class (by calling the SUPER pointer). The subclass has extended the base class method.
- **Example:**
 - The base class must control an axis for executing a process.
 - The subclass must also control an axis during process execution. **In addition**, the subclass must control a cylinder.
 - In this case, the method `ExecuteProcess` is created for the subclass. The implementation part of the method is programmed with the required additional instructions. The base class method (`FB_Base.ExecuteProcess`) is called at a suitable point in the sequence of the subclass method using the SUPER pointer (`SUPER^.ExecuteProcess (...)`). As a result, the subclass benefits from the implementations that have already been implemented in the base class. In addition, it can extend or customize the implementations with instructions that are required by the subclass.

Function block FB_Base:

```
FUNCTION_BLOCK FB_Base
VAR
    fbAxis : FB_Axis;
END_VAR
```

Method FB_Base.ExecuteProcess:

```
METHOD ExecuteProcess : BOOL
VAR_INPUT
    bExecuteProcess : BOOL;
END_VAR

// Calling axis module by passing input parameter "bExecuteProcess" of this method to the input
// parameter "bExecute" of method "Execute"
fbAxis.Execute(bExecute := bExecuteProcess);

// Setting the return value of this method as inverted error signal of the axis module
ExecuteProcess := NOT fbAxis.Error;
```

Function block FB_Sub:

```
FUNCTION_BLOCK FB_Sub EXTENDS FB_Base
VAR
    fbCylinder : FB_Cylinder;
END_VAR
```

Method FB_Sub.ExecuteProcess:

```
METHOD ExecuteProcess : BOOL
VAR_INPUT
    bExecuteProcess : BOOL;
END_VAR


// Extension: Calling cylinder module by passing input parameter "bExecuteProcess" of this method to
// the input parameter "bExecute" of method "Execute"
fbCylinder.Execute(bExecute := bExecuteProcess);

// Setting the return value of this method as inverted error signal of the cylinder module PLUS
// calling the base method and analyzing its return value
ExecuteProcess := NOT fbCylinder.Error AND SUPER^.ExecuteProcess(bExecuteProcess :=
bExecuteProcess);
```


7.20.6 Extending a structure

Like function blocks, structures can be extended. The structure then obtains the variables of the basic structure in addition to its own variables.

Creating a structure that extends another structure:

1. Select the PLC project object or a subfolder in the PLC project tree.
2. In the context menu select the command **Add > DUT...**
 - ⇒ The dialog **Add DUT** opens.
3. Enter a name and select **Structure** as the data type.
4. Select the **Extends** option and click the  button.
 - ⇒ The **Input Assistant** opens.
5. From the category **Data Unit types** select the structure to be extended by the new structure.
 - ⇒ The structure extends the basic structure.

Multiple inheritance is not allowed

i Multiple inheritance is not allowed for structures. It is not possible for a structure to extend more than one other structure.

- Not possible:
`TYPE ST_Sub EXTENDS ST_Base1, ST_Base2 :
 STRUCT
 ...`


See also:

- [Object DUT \[▶ 75\]](#)

7.20.7 Extending interfaces

Like function blocks, interfaces can be extended. The interface then obtains the interface methods and properties of the basic interface, in addition to its own.

Create an interface that extends another interface:

1. Select the PLC project object or a subfolder in the PLC project tree.
2. In the context menu select the command **Add > Interface...**
 - ⇒ The **Add Interface** dialog opens.
3. Enter a name for the new interface.
4. Select the **Advanced** option and click the  button.
5. The **Input Assistant** opens.
6. From the category **Interfaces** select the interface to be extended by the new interface.
 - ⇒ The interface extends the basic interface.

Multiple inheritance allowed

i Multiple inheritance is allowed for interfaces. It is possible that one interface extends more than one other interface.

- Possible:
`INTERFACE I_Sub EXTENDS I_Base1, I_Base2`

See also:

- [Object Interface \[▶ 185\]](#)

7.20.8 Implementation of an interface

The implementation of interfaces is based on the concept of object-oriented programming. Through common interfaces, you can use different but similar function blocks in a similar way.

A function block, which implements an interface, must contain all methods and properties that are defined in this interface (interface methods and interface properties). This means: The name, return type, inputs and outputs of the respective method or property must be exactly the same. If the element declarations in the interface and in the function block differ, or if the interface contains further elements, which are not included in the function block, the compiler reports an error.

For more information on the using an interface and an example, see section "[Object Interface \[► 185\]](#)"



When you create a new function block that implements an interface, TwinCAT automatically inserts all methods and properties of this interface under the new function block in the tree.



If you then add further methods or properties to the interface, TwinCAT does not automatically add these elements to the relevant function blocks. For the update, you must explicitly use the command **Implement interfaces**.

On executing this command the automatically created methods or properties will be provided with a pragma attribute that provokes compilation errors or warnings. This will support you in ensuring that automatically created elements do not inadvertently remain empty. For further information, please refer to the help page for the [Command Implement interfaces \[► 1063\]](#).

Implementing an interface in a new function block

- ✓ The currently open project has at least one interface object.
- 1. Select the PLC project object or a subfolder in the PLC project tree, and in the context select the command menu **Add > POU...**
 - ⇒ The **Add POU** dialog opens.
- 2. Enter a name for the new function block in the **Name** input field, for example "FB_SampleImp".
- 3. Select Function block.
- 4. Select **Implements** and click the  button.
- 5. In the input assistant, select the interface, for example "I_Itf1", from the **Interfaces** category and click **OK**.
- 6. To add another interface, click  again and select another interface.
- 7. Optionally you can select an **Access specifier** for the new function block from the selection list.
- 8. From the **Implementation language** selection list Select "Structured Text (ST)", for example.
- 9. Click on **Open**.
 - ⇒ TwinCAT adds the function block FB_SampleImp in the PLC project tree, and the editor opens. The first line says:
 FUNCTION_BLOCK FB_SampleImp IMPLEMENTS I_Itf1
 The methods and properties of the interface have now been inserted in the PLC project tree under the function block and you can now enter program code in the implementation section of the methods and properties.

Implementing an interface in an existing function block

- ✓ The currently open project has a function block, for example "FB_SampleImp", and at least one interface object, for example "I_Itf1".
- 1. In the PLC project tree double-click on the POU FB_SampleImp.
 - ⇒ The POU editor opens.
- 2. Extend the existing entry in the top line, FUNCTION_BLOCK FB_SampleImp, with IMPLEMENTS I_Itf1.
 - ⇒ The function block FB_SampleImp implements the interface I_Itf1.

3. Elements (methods or properties) that are defined in the interface, but not yet present in the function block, can be generated automatically in the function block by executing the command **Implement interfaces**, which is available in the context menu of the function block in the project tree.
 - ⇒ The methods and properties of the interface have now been inserted in the PLC project tree under the function block and you can now enter program code in the implementation section of the methods and properties.

See also:

- [Object Function block \[► 171\]](#)
- TC3 user interface documentation: [Command Implement interfaces \[► 1063\]](#)

7.20.9 Method call

To implement a method call, the actual parameters (arguments) are transferred to the interface variables. Alternatively, the parameter names can be omitted.

Depending on the declared access modifier, a method can be called in the following ways: only within its own namespace (INTERNAL), only within its own programming block and its derivatives (PROTECTED), or only within its own programming block (PRIVATE). With the PUBLIC option the method can be called anywhere.

Within the implementation a method can call itself recursively: either directly using the THIS pointer or using a local variable for the assigned function block.

Method call as virtual function call

Inheritance can result in virtual function calls. Virtual function calls allow the same call in a program source code to call different methods during runtime.

The method call is dynamically bound in the following cases:

- You call a method using a pointer to a function block (for example, pFB^.SampleMethod). In this situation, the pointer can point to instances of the type of the function block and to instances of all derived function blocks.
- You call a method of an interface variable (for example, iSample.SampleMethod). The interface can reference all instances of function blocks that implement this interface.
- A method calls another method of the same function block. In this case, the method can also call the method of an extended function block with the same name.
- A method is called via a reference to a function block. In this situation, the reference can point to instances of the type of the function block and to instances of all derived function blocks.
- You assign VAR_IN_OUT variables of a basic function block type to an instance of a derived FB type. In this situation, the variable can point to instances of the type of the function block and to instances of all derived function blocks.

Example

- The function blocks FB_Sub1 and FB_Sub2 each extend the function block FB_Base.
- FB_Base implements the interface I_Base, which defines the Method1 method.
- FB_Base and FB_Sub1 provide the method Method1. FB_Sub1 thus overwrites or extends the method of the base class FB_Base.
- FB_Sub2 does not provide the method. The function block continues to use the method of the base class FB_Base.
- In the MAIN program, an instance of the base class FB_Base, an instance of the subclass FB_Sub1 or an instance of the subclass FB_Sub2 is assigned to an interface variable and a reference to the base class. Which instance is assigned depends on the value of the variable nVar.
- The method Method1 is called via the interface variable and the reference variable. This method call is dynamic and can be executed for different instances (fbBase, fbSub1, fbSub2). The underlying method implementations differ between the instances.
 - The implementation of FB_Base.Method1 is executed for the instance fbBase.

- The implementation of FB_Sub1.Method1 is executed for instance fbSub1 because FB_Sub1 overwrites or extends the base class method.
- The implementation of FB_Base.Method1 is executed for the instance fbSub2, since the subclass FB_Sub2 neither overwrites nor extends the base class method, but uses it unchanged.

Interface I_Base with method Method1:

```
INTERFACE I_Base
METHOD Method1
```

Function block FB_Base with method Method1:

```
FUNCTION_BLOCK FB_Base IMPLEMENTS I_Base
METHOD Method1
```

Function block FB_Sub1 with method Method1:

```
FUNCTION_BLOCK FB_Sub1 EXTENDS FB_Base
METHOD Method1
```

Function block FB_Sub2 without own method:

```
FUNCTION_BLOCK FB_Sub2 EXTENDS FB_Base
```

Program MAIN:

```
PROGRAM MAIN
VAR
  nVar      : INT;
  fbBase    : FB_Base;
  fbSub1    : FB_Sub1;
  fbSub2    : FB_Sub2;
  iBase     : I_Base;
  refBase   : REFERENCE to FB_Base;
END_VAR

(* Choosing the desired instances via value of nVar:
  0 => fbBase
  1 => fbSub1
  2 => fbSub2 *)

IF nVar = 0 THEN
  iBase := fbBase;
  refBase REF= fbBase;

ELSIF nVar = 1 THEN
  iBase := fbSub1;
  refBase REF= fbSub1;

ELSIF nVar = 2 THEN
  iBase := fbSub2;
  refBase REF= fbSub2;
END_IF

// Regarding each of the following two calls via interface and via reference:
// If nVar is 0, FB_Base.Method1 will be called for instance fbBase
// If nVar is 1, FB_Sub1.Method1 will be called for instance fbSub1
// If nVar is 2, FB_Base.Method1 will be called for instance fbSub2

iBase.Method1();
refBase.Method1();
```

Additional outputs

According to the IEC 61131-3 standard, methods and normal functions can have additional outputs declared. When the method is called, variables are assigned to the additional outputs.

For more information see [Object Function](#) [▶ 81].

Calling a method recursively

NOTICE

Recursions are primarily used for processing recursive data types such as linked lists. Recursion should be used advisedly. An unexpectedly deep recursion can lead to stack overflow and thus to machine downtime.

Within its implementation, a method can call itself, either

- directly via the THIS pointer or
- indirectly using a local function block instance of the basic function block

Usually, a compiler warning is issued when such a recursive call is made. If the method has the pragma {attribute 'estimated-stack-usage' := '<estimated stack size in bytes>'}, the compiler warning is suppressed. An implementation example can be found in chapter [Attribute 'estimated-stack-usage' \[▶ 801\]](#).

See also:

- [Object Method \[▶ 173\]](#)
- [Object Function \[▶ 81\]](#)
- [Extending a function block \[▶ 191\]](#)
- Reference Programming: [SUPER \[▶ 692\]](#)
- Reference Programming: [THIS \[▶ 694\]](#)

7.20.10 ABSTRACT concept

The keyword ABSTRACT is available for function blocks, methods and properties. It enables the implementation of a PLC project with abstraction levels.

Abstraction is a key concept of object-oriented programming. Different abstraction levels contain general or specific implementation aspects.



Available from TC3.1 Build 4024

Application of the abstraction

It is useful to implement basic functions or commonalities of different classes in an abstract basic class. You implement specific aspects in non-abstract sub-classes.

The principle is similar to the use of an interface. Interfaces correspond to purely abstract classes that contain only abstract methods and properties. An abstract class can also contain non-abstract methods and properties.

Rules for the use of the keyword ABSTRACT

- Abstract function blocks cannot be instanced.
- Abstract function blocks can contain abstract and non-abstract methods and properties.
- Abstract methods or properties contain no implementation (only the declaration).
- If a function block contains an abstract method or property, it must itself be abstract.
- Abstract function blocks must be extended in order to be able to implement the abstract methods or properties.
- Hence: A derived FB must implement the methods/properties of its basic FB or it must also be defined as abstract.

Sample

Abstract basic class:

```
FUNCTION_BLOCK ABSTRACT FB_System_Base
```

The commonalities of all system modules are implemented in this abstract basic class. It contains the non-abstract property "nSystemID" and the abstract method "Execute" for this:

```
PROPERTY nSystemID : UINT
METHOD ABSTRACT Execute
```

whereas the implementation of "nSystemID" is the same for all systems, the implementation of the method "Execute" differs for the individual systems.

Non-abstract sub-class:

```
FUNCTION_BLOCK FB_StackSystem EXTENDS FB_System_Base
```

Non-abstract classes that are derived from the basic class are implemented for the specific systems. This subclass represents a stack. Since it is not abstract, it must implement the method "Execute" that defines the specific stack execution:

```
METHOD Execute
```

7.20.11 Samples

Basic OOP sample

Description	This PLC sample illustrates some of the basic functions of object-oriented programming (OOP). It features the following elements/functions: <ul style="list-style-type: none"> • Function blocks (FBs) • Methods • Properties • FB inheritance/extension • Interface (ITF) implementation and use
Sample project	https://infosys.beckhoff.com/content/1033/tc3_plc_intro/Resources/7644034443/.zip
Further information	In the documentation PLC: Object-oriented programming

Extended OOP sample

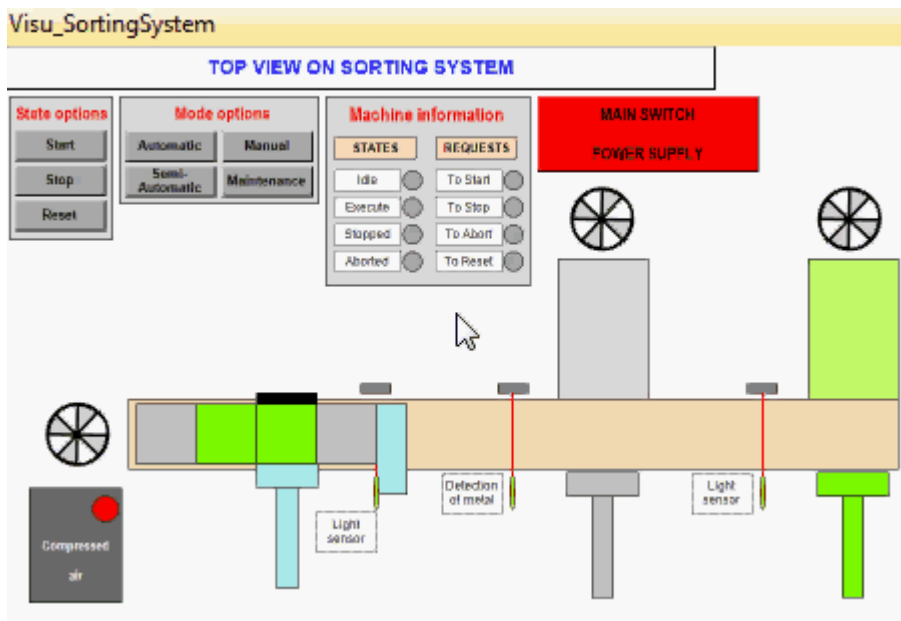
Description	This PLC sample contains an object-oriented program for controlling a sorting system. The application can be controlled via an integrated visualization.
Sample project	https://infosys.beckhoff.com/content/1033/tc3_plc_intro/Resources/7644036107/.zip
Further information	In the documentation PLC: Object-oriented program for controlling a sorting plant

7.20.11.1 Object-oriented program for controlling a sorting plant

Object-oriented programming offers a wide range of tools that can be used in many different ways. The following example illustrates the benefits of object orientation and demonstrates some ideas for using object orientation in PLC/machine programming. The object-oriented concept presented in this example and the underlying approaches do not claim to be exhaustive and cannot be generalized for other applications.

The example program for illustrating object-oriented programming (OOP) is used to control a sorting unit for metal and plastic boxes according to their material type. The sorting process includes one separation and two discharge units. These operations are implemented by means of cylinders, of which different versions are available. The different cylinder versions differ in terms of complexity. In the example, the system is required to be able to replace the cylinders in use with cylinders of a different type.

The system can distinguish the different material types by means of a sensor for metal detection, so that metal boxes (gray) and plastic boxes (green) are discharged on different auxiliary conveyor belts. The following video illustrates the sorting process, with the system viewed from above.

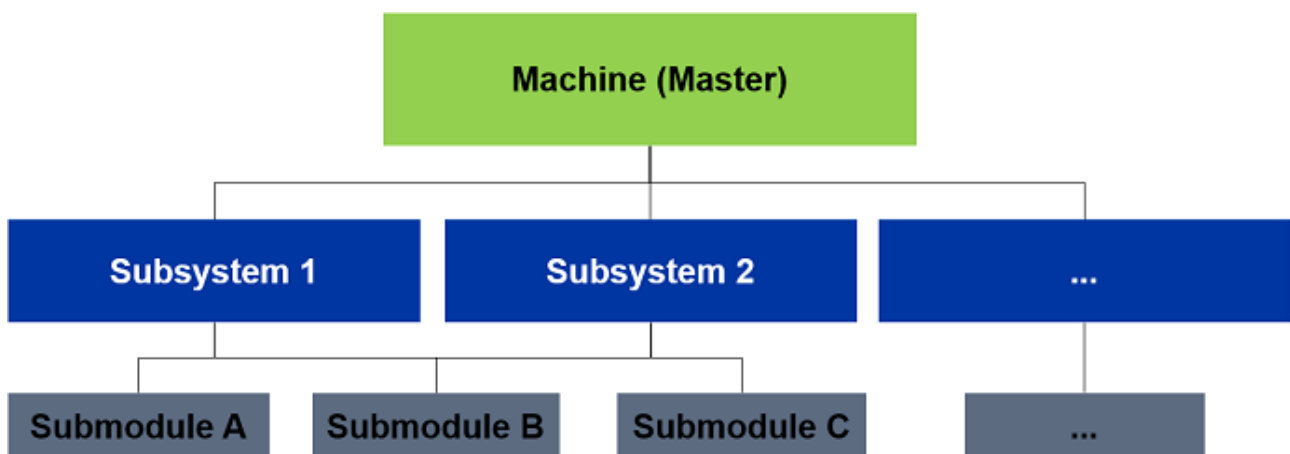


General concept of OOP

A general goal of object-oriented programming is to develop automation modules, for example. These objects are application-neutral, ready-made functional units, which only have to be developed once but can be used repeatedly without modification. Automation modules are therefore not developed for a specific plant, but generally provide system functionality that can be used in several plants. For example, a class “FB_Axis” for controlling an axis is developed once and then instantiated and used in several systems.

Automation modules can reduce the implementation effort for programming a system quite significantly. At the same time, the quality and reliability of the objects used are comparatively high, since the modules are also used in other systems and are therefore continuously tested and perhaps improved. If the same programming style and implementation concept are used for different modules, it is possible to achieve a uniform “look and feel” for the objects, so that application standardization can be achieved.

A prerequisite for meaningful development and application of automation modules is a modular design and modular programming of the system. For example, subdivision of the system into objects can be designed such that the system consists of different subsystems, and these subsystems contain different submodules. The machine, its subsystems and their submodules are implemented as automation modules and thus act as separate, independent objects.



Furthermore, it may be possible to generalize the data and functionalities that are common to all submodules in a submodule base class. By developing a subsystem base class, this approach can also be applied to subsystems, if there are commonalities between the subsystems. On the one hand, this would significantly reduce the programming effort, and on the other hand only the base class would have to be modified in the event of changes to the common implementations.

Sorting unit – approach

The programming of the sorting unit based on object orientation is explained in detail below, using the example of the cylinders. Implementing of the other submodules, such as drives and sensors, can be derived from the procedure for programming the cylinders. The consolidation of submodules into subsystems is briefly described at the end. Procedure for implementing the cylinders as submodules:

- Planning the software structure
- Implementing the software concept
- Instantiating the program elements
- Allocation of function block instances to an interface instance
- Use of a function block instance via an interface instance
- Further software concept: Consolidation of submodules to subsystems

Planning the software structure

The required cylinder types comprise simple cylinders, which can only move to a home or working position, and cylinders with additional functionalities. These more complex cylinders can monitor reaching of their end positions, and logging or diagnosing their temperature to detect whether the temperature is outside a specified range. The individual machine cylinders should not be limited to a certain cylinder type, but should be exchangeable with other types. In this way it should be possible, for example, to replace a simple cylinder with a cylinder with additional functionality.

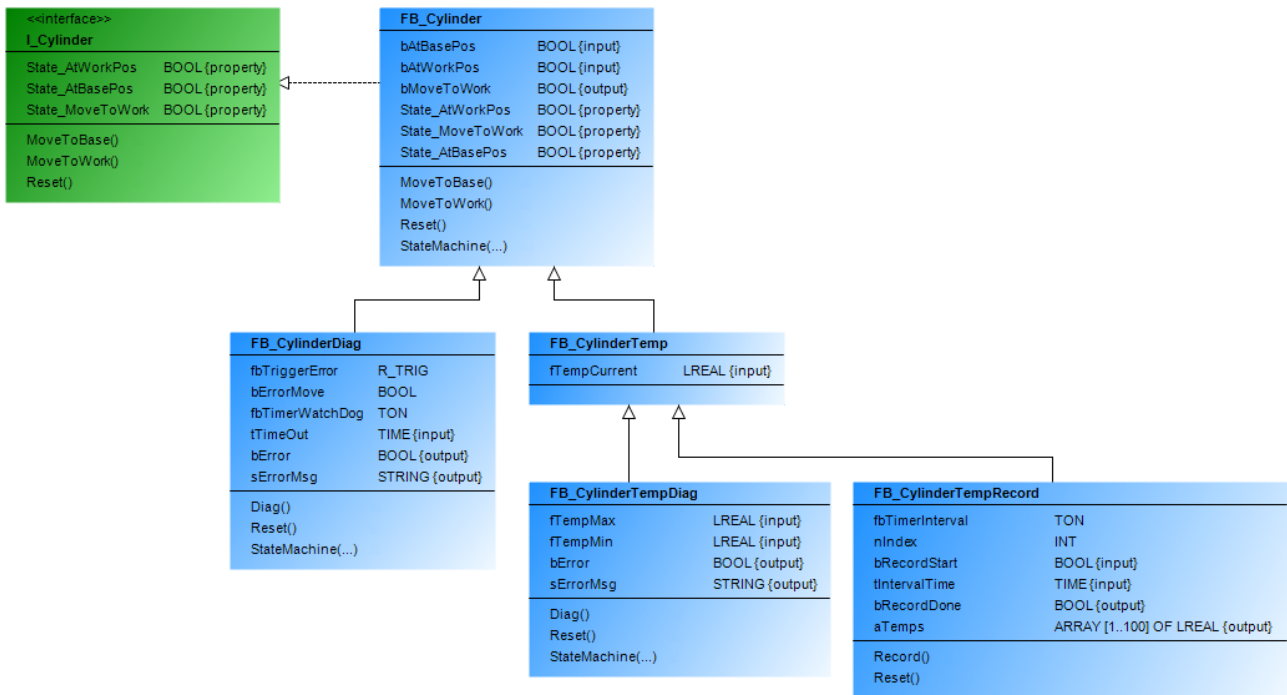
Irrespective of any auxiliary functions, all cylinders should be capable of the basic functionality of retracting and extending. Since this is a requirement that applies to each cylinder within the application, the methods and properties associated with the requirement are defined in an interface. An interface contains no implementations, only the definition of methods and properties. It therefore merely represents a convention for the function blocks that implement the interface. The convention is met if the defined methods and properties are provided by the function blocks. Finally, the implementations of the program elements defined in the interface are programmed in the implementing function blocks.

Defining the interface *I_Cylinder* and its implementation in the cylinder function blocks ensures that the cylinders meet the requirements and have access to the associated program elements. In addition, by defining and implementing an interface it is possible to access function block instances via interface instances. In this way the required cylinder exchangeability is ensured.

Since all cylinders have to provide the basic functionalities of retraction and extension, in addition to defining an interface convention it makes sense to generalize these functionalities and make their implementation available via a base class. For this reason the cylinder *FB_Cylinder* is planned, which features methods for cylinder movement and represents the base class for all required cylinders. This ensures that all derived classes offer the basic functionalities of retracting and extending, although they are only implemented once in the base cylinder.

The function block *FB_CylinderDiag* is derived from the superclass *FB_Cylinder*. As it also monitors reaching of the end positions, it extends the base cylinder with this functionality. It represents the solution for one of the required cylinders.

Since the cylinders require the value of the current temperature for recording and monitoring of the temperature, a function block with this component is derived from the base class *FB_Cylinder*. By adding a temperature supplement the class name becomes *FB_CylinderTemp*. It represents the superclass for cylinders with temperature recording and monitoring. The two subclasses with the names *FB_CylinderTempDiag* and *FB_CylinderTempRecord* are extended with specific behavior and variables required for these functionalities.



Implementing the software concept

The planned function blocks can now be implemented. In order to illustrate the procedure, parts of the base class *FB_Cylinder* and the derived class *FB_CylinderDiag* are described below.

The function block *FB_Cylinder* integrates the interface *I_Cylinder* with the aid of the keyword **IMPLEMENTS**, resulting in the methods and properties of the interface in the superclass to be created automatically. This ensures that the function block and its derivations have the elements required by the interface. The function block structure resulting from the interface and the declaration of the function block *FB_Cylinder* are shown in the diagram below.

```

1  (* FB_Cylinder    - number of control signals:
2                    one direction is controllable
3                    - type of feedback signal:
4                    feedback in base and work position    *)
5
6  FUNCTION_BLOCK FB_Cylinder IMPLEMENTS I_Cylinder
7  VAR_INPUT
8      bAtBasePos    AT %I*   : BOOL;    // Hardware input signal: cylinder is at base position
9      bAtWorkPos    AT %I*   : BOOL;    // Hardware input signal: cylinder is at work position
10 END_VAR
11 VAR_OUTPUT
12     bMoveToWork    AT %Q*   : BOOL;    // Hardware output signal to move cylinder to work position
13 END_VAR
    
```

The method *Reset*, which resets the cylinder output *bMoveToWork* of the function block, is shown as an example.

```

// =====
// *** Method Reset of FB_Cylinder ***
// =====
    bMoveToWork := FALSE;
    
```

With the aid of the keyword **EXTENDS** the function block *FB_CylinderDiag* becomes a derivative of *FB_Cylinder*. The variables, methods and properties of the base class can then be used (depending on the access modifier). Since the subclass is intended to extend the methods *Reset* and *StateMachine* of the base class, the methods are inserted in the subclass and can thus be modified. In addition, the method *Diag* is integrated, which is not included in the base class. The additional variables required for the diagnostic functionality are declared in *FB_CylinderDiag*. The structure and the declaration of *FB_CylinderDiag* are shown below:

```

1  (* FB_CylinderDiag - number of control signals:
2     one direction is controllable
3     - type of feedback signal:
4     feedback in base and work position
5     - with position diagnosis *)
6
7  FUNCTION_BLOCK FB_CylinderDiag EXTENDS FB_Cylinder
8  VAR_INPUT
9     tTimeout      : TIME;      // Time for watchdog that monitors if cylinder reaches base/work position
10 END_VAR
11 VAR_OUTPUT
12     bError        : BOOL;      // Error signal (diagnosed from position watchdog)
13     sErrorMsg     : STRING;    // Error message
14 END_VAR
15 VAR
16     fbTriggerError : R_TRIG;   // Trigger to recognize rising edge of error
17     bErrorMove     : BOOL;     // Move error
18     fbTimerWatchDog : TON;     // Watchdog timer for monitoring if cylinder reaches base/work position
19 END_VAR

```

In the methods *Reset* and *StateMachine* of the derived class, the corresponding methods of the superclass can be extended or overwritten. To obtain a method extension the method of the base class is called by using the keyword *SUPER*. *SUPER* is a function block pointer that points to the function block instance of the base class. The behavior of the base class can be extended through further instructions in the method of the subclass, whereby the method is adjusted for the cylinder *FB_CylinderDiag*. As an example, the method *Reset* of *FB_CylinderDiag*, which extends the corresponding method of the base class *FB_Cylinder* through further instructions, is shown below.

```

// =====
// *** Method Reset of FB_CylinderDiag ***
//
// Calling method Reset of base class FB_Cylinder via 'SUPER^.'
SUPER^.Reset();
//
// Reset error
bError      := FALSE;
sErrorMsg   := '';
// =====

```

Instantiating the program elements

The function blocks created with the aid of inheritance and corresponding keywords, which represent cylinders with different functionality, can now be instantiated.

The function blocks *FB_Cylinder*, *FB_CylinderDiag*, *FB_CylinderTemp*, *FB_CylinderTempDiag* and *FB_CylinderTempRecord* are instantiated once to ensure that a cylinder can be regarded as variable and can be represented by each cylinder type. The interface instance *iCylinder* is the object that references one of the cylinder function block instances and therefore the currently selected cylinder type at runtime. The variables *bCylinderDiag*, *bCylinderTemp* and *bCylinderRecord* can be modified by the user and indicate whether the cylinder has diagnostics and temperature functionality.

The instantiation of the function blocks, the interfaces and the three Boolean variables are shown below.

```

// ===== Variables to enable/disable diagnosis and temperature mode =====
bCylinderDiag      : BOOL;      // If true the cylinder has diagnosis
functionality
bCylinderTemp      : BOOL;      // If true the cylinder has temperature
functionality
bCylinderRecord    : BOOL;      // If true the cylinder has recording
functionality

// ===== Function block instances for cylinder =====
fbCylinder          : FB_Cylinder;      // Without diagnosis and temperature mode
fbCylinderDiag      : FB_CylinderDiag;  // With diagnosis of states
fbCylinderTemp      : FB_CylinderTemp;  // With temperature mode
fbCylinderTempDiag  : FB_CylinderTempDiag; // With diagnosis of temperature
fbCylinderTempRecord : FB_CylinderTempRecord; // With record of temperatures

// ===== Interface instance for cylinder =====
iCylinder           : I_Cylinder;      // Interface for flexible access to cylinder FBs

```

Assignment of function block instances to an interface instance

To exchange a cylinder all that is required is to adapt the variables *bCylinderDiag*, *bCylinderTemp* and *bCylinderRecord*, since the corresponding function block instance is assigned to the interface instance *iCylinder*, depending on the state of these three variables.

If the cylinder is to have the functionality of temperature monitoring, for example, *bCylinderDiag* and *bCylinderTemp* have the value *TRUE* and *bCylinderRecord* has the value *FALSE*. The desired function block

is therefore *FB_CylinderTempDiag* and the associated instance *fbCylinderTempDiag* is assigned to the interface instance. The specific outputs *bError* and *sErrorMsg* for this class are intercepted separately by assigning them to local variables. Because the FB instance *fbCylinderTempDiag* is assigned to the interface instance *iCylinder*, the function block instance with temperature monitoring can be accessed via the interface instance.

If the Boolean variables have other values, conversely, the interface instance will be assigned a different function block instance accordingly. Two samples for function block assignments are shown below.

```
// =====
// Selecting cylinder by checking variables to enable / disable diagnosis and temperature mode

IF bCylinderDiag THEN
  IF bCylinderTemp THEN
    // ===== FB with diagnosis and temperature mode =====
    bError      := fbCylinderTempDiag.bError;          // Assigning output variable of
chosen FB to local variable
    sErrorMsg   := fbCylinderTempDiag.sErrorMsg;
    iCylinder   := fbCylinderTempDiag;                // Assigning chosen FB instance to
interface instance

  ELSE
    // ===== FB with diagnosis and without temperature mode =====
    fbCylinderDiag.tTimeout := tTimeoutCylinder;      // Setting special data for
selected FB
    bCylError    := fbCylinderDiag.bError;          // Assigning output variable of
chosen FB to local variable
    sCylErrorMsg := fbCylinderDiag.sErrorMsg;
    iCylinder    := fbCylinderDiag;                // Assigning chosen FB instance to
interface instance

    ...

  END_IF
// =====
```

Use of a function block instance via an interface instance

The interface instance that was assigned one of the FB instances can call its methods with the help of dot notation. With this method call via the interface instance, the method of the function block instance to which the interface points is called due to the interface pointer.

The following program section illustrates how the selected cylinder function block is moved to the home or working position via the interface instance.

```
// =====
// Manual cylinder control (Calling methods of FB via interface instance)

// Cylinder to work position
IF fbButtonCylToWork.bOut THEN
  iCylinder.MoveToWork();
// Cylinder to base position
ELSIF fbButtonCylToBase.bOut THEN
  iCylinder.MoveToBase();
END_IF
// =====
```

Further software concept: Consolidation of submodules to subsystems

The implementation of function blocks and the application of their instances with the aid of an interface instance presented here represents small-scale object-oriented programming. For optimum application of object orientation it is complemented with larger-scale OOP. Different submodules are consolidated to subsystems.

For a sorting unit, it makes sense to consolidate a sensor for material detection, a drive for controlling the movement of an auxiliary conveyor belt and a discharge cylinder to form a discharge module. In this way it is possible to instantiate any number of objects or sorting modules from this class, with only one discharge module being programmed. Based on this approach it is possible to configure different systems with different numbers of sorting modules that are specialized for other material type (e.g. metal, plastic, foil, glass etc.). Another subsystem of the sorting unit deals with separation. It consists of a sensor for box identification, a drive for moving the boxes to the main conveyor and two cylinders for implementing the actual separation. Since the two subsystems for separation and discharging have commonalities, the subsystem-specific data

and functionalities are consolidated in a subsystem base class. Separation and discharging are then declared as a subclass of this function block, which means they inherit the generalized program elements from the subsystem base class.

Through the development of submodules and subsystems, the object-oriented programming options and benefits can be applied and utilized at various levels.

TC3.1 sources

The complete TC3.1 sources for the sample program can be unpacked here: https://infosys.beckhoff.com/content/1033/tc3_plc_intro/Resources/7644036107/.zip

To start the sample program:

- Activate the TwinCAT configuration and start TwinCAT in Run mode
- Log in on both PLCs and start them (**TC3_SortingSystem_PLC** and **TC3_SortingSystem_Simu**)
- Operate the application via the machine visualization. This can be found in:
 - PLC project: TC3_SortingSystem_PLC
 - Folder: 05_Visu
- For example: Start the system in automatic mode by pressing the following button:
 - “Main Switch Power Supply”
 - “Automatic”
 - “Start”

See also:

- PLC documentation: [Object-oriented programming](#)

8 Transfer PLC project to the PLC

In order to transfer the PLC project to the controller, the program must be compiled without errors.

8.1 Generating program code

The program code is the machine code, which a controller executes when you start a PLC program. TwinCAT automatically generates the program code from the source code written in the development system before downloading the PLC project to the controller. Before the program code is generated, the system checks the assignments, data types and availability of libraries. In addition, the memory addresses are allocated when the program code is generated.

For each download, the compile log (Compile Info), which contains the code and identification of the loaded PLC project, is saved as a file on the target device.





Messages during program code generation

Due to the incremental compilation, the memory is only reassigned for new and modified function blocks and variables. It is therefore possible that gaps occur in the memory. Online changes have the same effect. This fragmentation reduces the available free memory. In this case, you can use the command Clean to re-allocate the entire memory, thus increasing the free memory. Further information on messages during code generation: syntax errors and errors detected by TwinCAT during the code generation and memory allocation appear in the message window Compile (error list). During each code generation, information about the size of the code, the size of the data (in bytes), the content of the occupied memory areas, and the highest address used (byte) is also displayed there. It depends on the PLC in which memory areas which data and the code are stored.

8.2 Loading program code, logging in and starting the PLC

To download the source code of your PLC program onto the controller, you have to log in with the PLC project on the controller. If you have several PLC projects in your project, you have to activate the desired PLC project first.

Loading the PLC project and starting the program

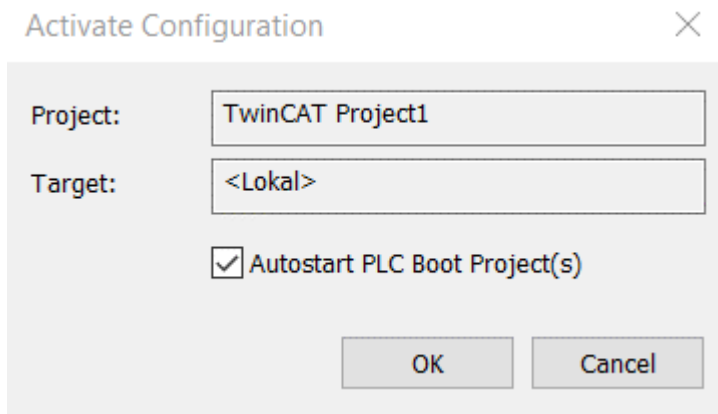
- ✓ The PLC project is error-free and is not yet on the controller.
- ✓ The PLC project and the communication with the controller are not encrypted.
- 1. Select the PLC project to be the loaded and started in the drop-down list **Active PLC Project** in the **TwinCAT PLC Toolbar Options**.
 - ⇒ The active PLC project appears as the first entry in the drop-down list.
- 2. In the **TwinCAT XAE Base Toolbar Options** click on **Activate Configuration**  .
 - ⇒ A dialog appears asking whether you want to activate the configuration.
- 3. Click **OK**.
 - ⇒ A dialog appears asking whether TwinCAT should be restarted in Run mode.
- 4. Click **OK**.
 - ⇒ The configuration is activated, and TwinCAT is set to Run mode. The current status appears in the taskbar:  . Activation also transfers the PLC project to the controller.
- 5. Select the command **Login** in the **PLC** menu or in the **TwinCAT PLC Toolbar Options**  .
 - ⇒ A dialog appears asking whether you want to create and load the application onto the controller.
- 6. Confirm the dialog with **YES**.
 - ⇒ The PLC project is loaded onto the controller.
- 7. Select the command **Start** in the **PLC** menu or in the **TwinCAT PLC Toolbar Options**  or press **[F5]**.
 - ⇒ The program is running on the controller.

See also:

- [Updating the PLC project on the PLC \[► 250\]](#)
- TC3 User Interface documentation: [Command Login \[► 957\]](#)
- TC3 User Interface documentation: [Command Activate configuration \[► 937\]](#)
- [Monitoring values](#)
- [Programming languages and their editors](#)

8.3 Loading the program automatically

If you activate the configuration in your PLC project  , a dialog box opens where you can see or set whether the Autostart PLC Boot Project(s) is (are) activated.



The checkbox in the dialog indicates whether or not the autostart is currently activated for PLCs on the target:

- Checked: All autostarts on the target are set.
- Unchecked: All autostarts on the target are removed.

To change the autostart function for your PLCs, set or remove the checkmark in the checkbox. When you confirm your selection with **OK**, your entry will be saved and you will be taken to the TwinCAT Run Mode.

The option corresponds to the **Autostart Boot Project** command from the **Project** tab of the PLC project in the Solution Explorer.

9 Testing a PLC project and troubleshooting

TwinCAT 3 PLC offers you various options for testing your application and finding errors. You can start your application in simulation mode even without hardware being connected. With breakpoints and commands for executing the program step by step, you can examine specific parts of the program. You can influence the running program by writing variable values.

Commands are available to you that reset your application to varying degrees. These extend from the resetting of non-persistent variables to the complete resetting of the controller to the delivery state.

9.1 Use of breakpoints











Breakpoints are generally used for finding errors in the program. You can set breakpoints at certain positions in the program in order to force an execution stop there and observe the variable values. TwinCAT 3 PLC supports breakpoints in all IEC editors.

The stop at the breakpoint can be linked to additional conditions. You can also redefine breakpoints as execution points at which the program doesn't stop, but instead executes certain code.



The view **Breakpoints** (menu **PLC > Windows**) provides an overview of all defined breakpoints. Additional commands are available to you there for the simultaneous changing of several breakpoints.

The status of breakpoints and execution points is marked in the editor with the following symbols:

-  Breakpoint activated
-  Breakpoint deactivated
-  Breakpoint is set in a different instance of the function block currently open in the editor.
-  Stop at breakpoint
-  Breakpoint with conditions activated
-  Breakpoint with conditions deactivated
-  Execution point activated
-  Execution point deactivated
-  Execution point with condition activated
-  Execution point with condition deactivated

See also:

- TC3 User Interface documentation: [PLC \[► 939\]](#)
- TC3 User Interface documentation: [Debug \[► 930\]](#)
- TC3 User Interface documentation: [Command Call Stack \[► 945\]](#)

Breakpoints in PLC projects with several tasks

If a breakpoint is reached when executing a PLC project, no further code in this PLC project will be executed by any task. Code located outside of this PLC project will still be executed.





If the program on the PLC has stopped at a breakpoint, an online change or download stops all tasks. This means that the PLC stops. In this case TwinCAT displays a corresponding message and you can decide whether or not you wish to continue with the login.

Setting a single breakpoint (example: ST editor)

✓ The project is in online mode.

1. Open a POU in ST language in the editor.
2. Place the cursor in the line in which a breakpoint is to be set.
3. Select the command **Toggle Breakpoint** in the menu **Debug** or in the context menu or press the **[F9]** key.

⇒ The line is colored red and marked with the icon  (breakpoint enabled). If the program has stopped at the breakpoint, the line is marked with the icon  (stop at breakpoint). The execution of the program stops.

4. Select the command **Start** in the menu **PLC** or in the toolbar **TwinCAT PLC Toolbar Options** or press the **[F5]** key.

⇒ The program continues to run.

5. Set further breakpoints and check the values of variables at the stopping positions.
6. Place the cursor in the line in which a breakpoint is to be deleted.
7. Select the command **Toggle Breakpoint** in the menu **Debug** or in the context menu or press the **[F9]** key.

⇒ The marking disappears. The breakpoint is deleted.

See also:

- TC3 User Interface documentation: [Command Toggle Breakpoint \[► 934\]](#)

Defining a breakpoint condition (example: ST editor)

✓ The project is in online mode.

1. Open a POU in ST language in the editor.
2. Select the command **Breakpoints** in the menu **PLC > Windows**.

⇒ The **Breakpoints** view opens.

3. Select the command **New** in the toolbar.

⇒ The dialog **New Breakpoint** opens. The tab **Location** is visible. Alternatively, you can open the dialog using the command **New Breakpoint** in the menu **Debug**.

4. Select the POU and the position of the new breakpoint.
5. Select the tab **Condition**.
6. In the section **Hit Count**, select the option **Break when the hit count is a multiple of** and enter the value 5 in the field to the right of it.
7. Also define a Boolean condition for when the breakpoint should be active. Activate the option **Break if TRUE**. Enter a Boolean variable in the field to the right of it.
8. Activate the option **Enable breakpoint immediately**.
9. Close the dialogue.

⇒ The line is colored red and marked with the icon .

Now observe the running program. As long as the Boolean variable for the condition is FALSE, the condition for the breakpoint is not satisfied and the program runs. If you set the variable to TRUE, the condition is satisfied and the program stops at this breakpoint every 5th pass.


See also:

- TC3 User Interface documentation: [Command Breakpoints \[► 943\]](#)
- TC3 User Interface documentation: [Command New Breakpoint \[► 930\]](#)

Defining an execution point (example: ST editor)

✓ The project is in online mode.

1. Open a POU in ST language in the editor.


2. From the menu **PLC > Window** select the command **Breakpoints**.
 - ⇒ The **Breakpoints** view opens.
3. Select the command **New** in the toolbar.
 - ⇒ The dialog **New Breakpoint** opens. The **Location** tab is visible.
Alternatively, you can open the dialog using the **New Breakpoint** command in the **Debug** menu.
4. Select the POU and the position of the execution point.
5. Select the **Execution point settings** tab.
6. Activate the **Execution point** option.
 - In the field **Execute the following code**, enter the desired instructions that are to be executed on reaching the execution point. For example, `nCounter := nCounter + 1;`, if the variable `nCounter` is available.
7. Close the dialog.
 - ⇒ The line is colored red and marked with the icon 

The program does not stop on reaching the execution point; instead, the defined code is executed.


See also:

- TC3 User Interface documentation: [Command Breakpoints \[► 943\]](#)
- TC3 User Interface documentation: [Command New Breakpoint \[► 930\]](#)



9.2 Stepwise processing of the program (stepping)

You can execute a PLC project step by step, navigating through the code. This is useful for determining the status of your code at runtime. You can examine the call sequence, track variable values or determine errors. For this purpose, step commands are available in the menu **Debug**. The commands become available when the program is at a defined program step (debug mode). During debug mode, the current stop position is highlighted in yellow and marked in the text editors with the symbol .

Switch to debug mode

- ✓ The PLC project is in online mode.
- 1. Set breakpoints in the POU's at the locations in the code you want to examine.
- 2. Start the program.
 - ⇒ The program starts, the code is processed to the first breakpoint. The project is now in debug mode.
 - ⇒ The editor with the current stop position is opened. The line of code with an active breakpoint, where the program execution was stopped, is highlighted in yellow and marked with the symbol  (stop at breakpoint). The instruction has not been executed yet.
 - ⇒ You can select the different step commands or display the call stack.

Behavior of the step commands

- [Command Step over \[► 935\]](#) ("Step Over") 
 - The instruction at the stop position is executed. The program stops before the next instruction in the programming block.
 - If there is a call in the statement (from a program, a function block instance, a function, a method or an action), the subordinate programming block is completely traversed in one step.
- [Command Step into \[► 935\]](#) ("Step Into") 
 - The instruction at the stop position is executed. The program stops before the next instruction.

If there is a call in the instruction (from a program, a function block instance, a function, a method or an action), the program jumps to this subordinate programming block. The first instruction there is executed and the program is stopped before the next instruction. The new current stop position is then in the called programming block.

- [Command Step out \[▶ 935\]](#) ("Step Out") 

The command executes the programming block from the current stop position to the end of the block and then jumps back to the calling programming block. At the call position (in the line with the call) the program is stopped.

If the current stop position is in the main program, the programming block is run through to the end. Then the program jumps back to the beginning (to the program start at the first code line in the programming block) and stops there.

- [Command Run To Cursor \[▶ 936\]](#) ("Run to Cursor") 

First place the cursor at any line of code and then select the command. The program is executed from the current stop position and stops at the current cursor position without executing the code of this line.

- [Command Set next statement \[▶ 936\]](#) ("Set Next Statement") 

First place the cursor at any line of code (even before the current stop position) and then select the command. The instruction marked with the cursor will be executed next. All instructions in between are ignored and skipped.

- [Command Show Next Statement \[▶ 936\]](#) ("Show Next Statement") 






If you do not see the current stop position, select the command. Then the window with the current stop position becomes active and the stop position becomes visible.

Select the command PLC > Window > Call Stack to display the previous call tree completely for the stop position currently reached in the program execution.



The **Call Tree** view therefore shows where the function blocks are located in the call structure of the program at all times, even before compilation of the PLC project.

Also see about this

-  [Command Step into \[▶ 935\]](#)
-  [Command Step over \[▶ 935\]](#)
-  [Command Step out \[▶ 935\]](#)
-  [Command Run To Cursor \[▶ 936\]](#)
-  [Command Show Next Statement \[▶ 936\]](#)

9.3 Forcing and Writing Variables Values


In TwinCAT you can change the values of variables on the controller in online mode. This overwrites the originally entered value of the variable. There are two different approaches to this, forcing and writing a previously prepared value.

CAUTION

Damage to property and persons due to unexpected behavior of the machine or system

The abnormal changing of variable values in a PLC program that is running on the controller can result in unexpected behavior of the controlled machine. Depending on the machine being controlled, the machine or workpieces may be damaged or the health and life of people may be endangered.

- Evaluate possible risks before forcing variable values and take appropriate safety precautions.

Writing is done with the command **Write values**  and sets the variable once to the prepared value. The value can thus be overwritten by the program at any time.

Forcing is done with the command **Force values**  and sets the prepared value permanently.

Preparing a value for forcing or writing is possible at various points:

- Declaration part: Field **Prepared value**
- Implementation part: Inline monitoring field
- Monitoring window: Field **Prepared value**

Operating principle of Forcing

In the case of forcing, TwinCAT writes the value in each cycle so that the variable is permanently kept at the forced value. Forcing must be canceled by the user. The forced value of the variable is changeable within a PLC cycle, as with any other variable.

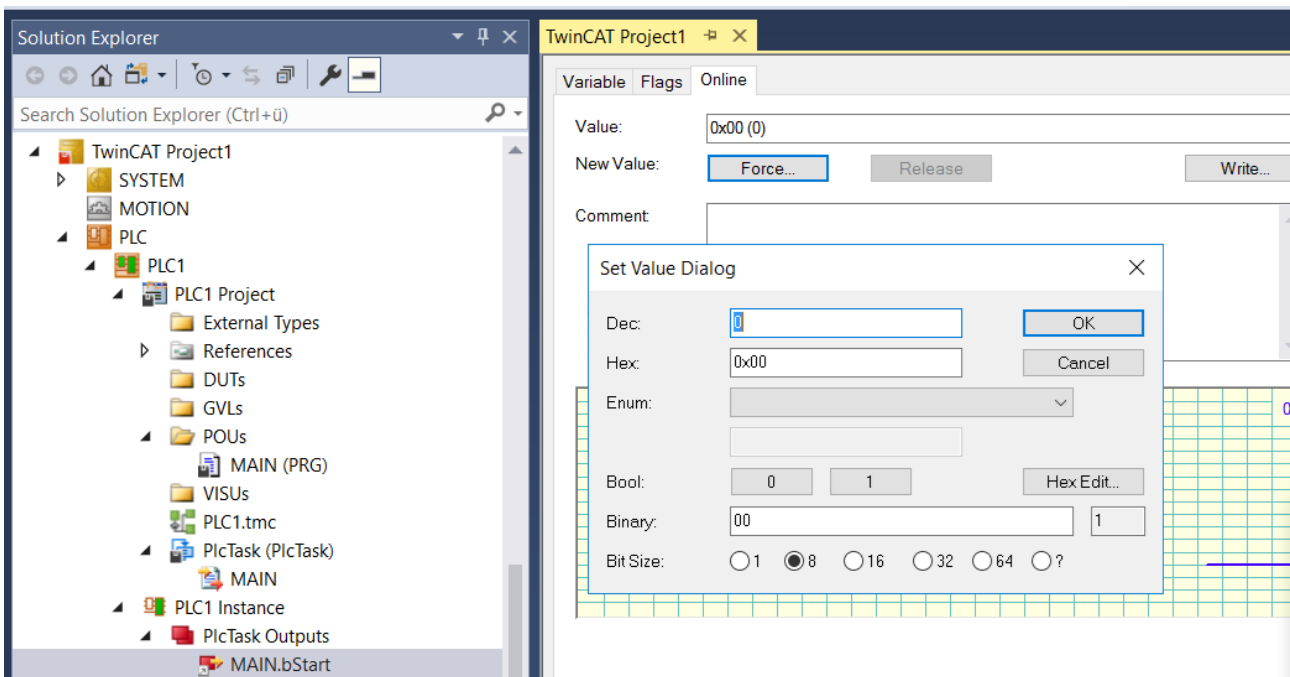
Setting the prepared value to the respective variable is done at the beginning and end of each processing cycle. Sequence of processing in each cycle:

1. Read inputs
2. Force values
3. Process code
4. Force values
5. Write outputs.

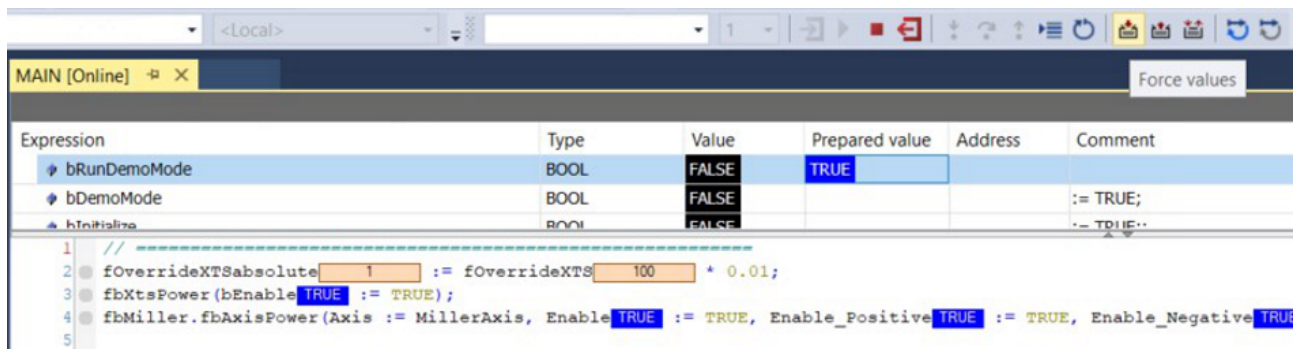
It is possible that a forced variable temporarily gets a different value during code processing in the cycle because the code performs an assignment. The variable then receives the forced value again only at the end of the cycle. Also by the write access of a client to symbols of the application the variable value can be overwritten during the cycle.

There are two different approaches to the forcing of values.

For one approach the project can be opened from the Solution Explorer. The values are then forced directly from here. Logging in to the runtime is not necessary for this. This keeps the value of the variable forced even if the user leaves the runtime system. In this case, neither a warning message nor a query dialog will appear.



In contrast, it is only possible to force a variable in the PLC if the user logs into the runtime system .




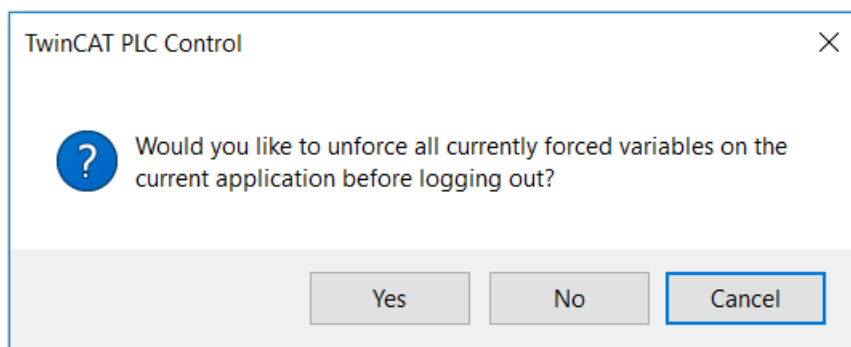
Expression	Type	Value	Prepared value	Address	Comment
bRunDemoMode	BOOL	FALSE	TRUE		
bDemoMode	BOOL	FALSE			:= TRUE;
bInitialize	BOOL	FALSE			:= TRUE;

```

1 //
2 fOverrideXTSabsolute 1 := fOverrideXTS 100 * 0.01;
3 fbXtsPower(bEnable TRUE := TRUE);
4 fbMiller.fbAxisPower(Axis := MillerAxis, Enable TRUE := TRUE, Enable_Positive TRUE := TRUE, Enable_Negative TRUE := TRUE);
5

```

When logging out  of the runtime system, a dialog appears asking whether the variable should remain forced.



If you select **Yes** in this dialog, the forced values will be canceled. If you select **No** here, the forced values are stored on the runtime system and accordingly remain permanent. This means that you can log out of the runtime system in the meantime and the forced values still exist if you log back into the runtime system at another time.


NOTICE

Material damages due to permanently forced variables

Variables are held permanently at the forced value, as a result of which the forced value can persist longer than expected and material damage can occur. This applies in particular if the machine is running without supervision.

- To ensure that the machine does not perform any unexpected movements, reset forced values at the end of the machining process.



Please note that forced variables  have to be canceled explicitly by the user. However, the forced value of the variable can persist even after unforcing.

- Canceling the forcing of the variable.
- To securely cancel the forced value of the variable, change the values back to the original values.
- Log out of TwinCAT and confirm the query whether forcing should be canceled for all variables with **Yes**.

⇒ The variable now has its original value again.

Temporary variables cannot be forced. Temporary variables can only be written if the PLC is at a breakpoint that is located in the code of the same POU in which the temporary variable is defined. The writing of temporary variables also has no effect in flow control mode.


See also:

- TC3 User Interface documentation: [Command Force values \[► 960\]](#)
- TC3 User Interface documentation: [Command Write values \[► 963\]](#)
- TC3 User Interface documentation: [Command Unforce values \[► 962\]](#)

Forcing in the declaration part

- ✓ Your PLC project has a POU with declarations. The application is in online mode.
- 1. Open the POU in the editor by double-clicking on the object or on the command **Open** in the menu **View** or the context menu.
- 2. In the declaration part of the editor, double-click in the column **Prepared value** of a variable.
 - ⇒ The field becomes editable and you can enter a value.

TwinCAT_Device.Project12.MAIN						
Expression	Type	Value	Prepared value	Address	Comment	
fbColors	FB_Colors					
nColorR	INT	0	100			
nColorY	INT	0	200			
nColorG	INT	0	300			

- 3. Execute step 2 for further variables.
- 4. Select the **Force values** command  in the **PLC** menu or in the **TwinCAT PLC toolbar options** toolbar.
 - ⇒ The values of the variables are overwritten by the prepared values. The values are marked with the symbol **F**.

TwinCAT_Device.Project12.MAIN						
Expression	Type	Value	Prepared value	Address	Comment	
fbColors	FB_Colors					
nColorR	INT	F 100				
nColorY	INT	F 200				
nColorG	INT	F 300				



You can also force values of variables in the view **PLC > Windows > Watch <n>**.

See also:

- TC3 User Interface documentation: [Command Force values \[► 960\]](#)

Forcing in the implementation part

- ✓ The application is in online mode.
- 1. Open the POU in the editor by double-clicking on the object or on the command **Open** in the menu **View** or the context menu.
- 2. Double-click on an inline monitoring field in the implementation part of the editor.
 - ⇒ The dialog **Prepare Value** opens.
- 3. In the field **Prepare a new value for the next write or force operation**, enter the new value.
 - ⇒ The prepared value appears in the inline monitoring field.

```

5 nColorR1 0 := nColorG 0 <100> / 100;
6 RETURN
    
```

- 4. Select the command **Force values** in the menu **PLC** or in the toolbar **TwinCAT PLC Toolbar Options**.
 - ⇒ The value of the variable is overwritten by the prepared values. The values are marked with the symbol **F**.

```

5 nColorR1 1 := nColorG F 100 / 100;
6 RETURN
    
```

See also:

- TC3 User Interface documentation: [Dialog Prepare Value \[► 961\]](#)
- TC3 User Interface documentation: [Command Force values \[► 960\]](#)
- TC3 User Interface documentation: [Command Unforce values \[► 962\]](#)

Viewing and canceling all forced variables in a list

✓ The application is in online mode. Several variables are in the forced state.

1. Select the command **Watch all forces** in the menu **PLC > Windows**.

⇒ The view **Watch all forces** appears. It contains all currently forced variables of the PLC project in the form of a watch list.

2. Select all rows of the list and, in the selection list at the top left select in the view **Unforce > Unforce and keep all selected values**.

⇒ Forcing is canceled for the variables and they are given the values that they had before forcing.

See also:

- TC3 User Interface documentation: [Command Watch all forces \[► 940\]](#)
- TC3 User Interface documentation: [Command Write values \[► 963\]](#)
- TC3 User Interface documentation: [Command Unforce values \[► 962\]](#)

Also see about this

- 📖 [Command Write values \[► 963\]](#)
- 📖 [Command Force values \[► 960\]](#)
- 📖 [Command Unforce values \[► 962\]](#)

9.4 Resetting the PLC project

A reset of the PLC project stops the program and resets the variables to their initialization value. RETAIN variables and PERSISTENT variables are also reset depending on the type of reset.

- Reset cold: All variables of the active PLC project, with the exception of the remanent variables (RETAIN and PERSISTENT variables), are reset to their initialization value.
- Reset original: All variables of the active PLC project, including the remanent variables (RETAIN and PERSISTENT variables), are reset to their initialization value. The PLC project on the controller is reset.

The small sample program and the following instructions illustrate the behavior of the different resets to you.

See also:

- Reference Programming: [Remanent Variables - PERSISTENT, RETAIN \[► 691\]](#)
- TC3 User Interface documentation: [Command Reset cold \[► 959\]](#)
- TC3 User Interface documentation: [Command Reset origin \[► 959\]](#)

Sample program

Declaration:

```
VAR
  nVar : INT := 0;
END_VAR
VAR RETAIN
  nVarRetain : INT :=0;
END_VAR
VAR PERSISTENT
  nVarPersistent : INT:= 0;
END_VAR
```

Implementation:

```
nVar := 100;
nVarRetain := 200;
nVarPersistent := 300;
```

1. Execute the command **Build**.
 2. Load the PLC project into the controller.
 3. In the menu **PLC** or in the toolbar **TwinCAT PLC Toolbar Options**, select the command Login to switch to online mode.
 4. Start the PLC program.
- ⇒ Observe the variables nVar, nVarRetain and nVarPersistent.

Performing a Reset cold:

1. Select the command **Reset cold** in the menu **PLC** or in the **TwinCAT PLC Toolbar Options**.
 - ⇒ A query appears, asking if you really want to execute the command.
 2. Confirm the dialog with **Yes**.
- ⇒ The PLC project is reset. The variable nVar is set to the initialization value 0. The RETAIN variable nVarRetain and the PERSISTENT variable nVarPersistent retain their value

Performing a Reset origin:

1. Select the command **Reset origin** in the menu **PLC** or in the **TwinCAT PLC Toolbar Options**.
 - ⇒ A query appears, asking if you really want to execute the command.
 2. Confirm the dialog with **Yes**.
- ⇒ The PLC project is logged out. All variables are reset to their initialization value.

9.5 Flow Control

You can track the execution of the program with flow control. The flow control is available for the language editors ST, FBD, LD and CFC.

With flow control activated, TwinCAT displays the values of variables and the results of function calls and operations at the respective execution position and at the respective execution time. Precisely the code lines or networks being run through in the current cycle are marked in color. For comparison: with standard monitoring, TwinCAT returns only the value that a variable has between two execution cycles.

The flow control operates in all currently visible parts of the currently opened editor window. **Flow Control activated** is thereby displayed in the status bar as long as the function is active and flow control positions (parts of the code that have been run through) are visible in an editor window.

You can write values in the declaration part and in the implementation part. Forcing is not possible.



Writing of the values is done at the end of the current cycle.



The runtime of the PLC project is extended if you activate flow control.

Representation of the flow control in various language editors

By default, TwinCAT represents the flow control positions of the code parts that have been run through as green fields. Code parts that have not been run through are displayed in white.



Please note that the displayed value of a code position that has not been run through is a "normal" monitoring value. This is the value that exists between two task cycles.

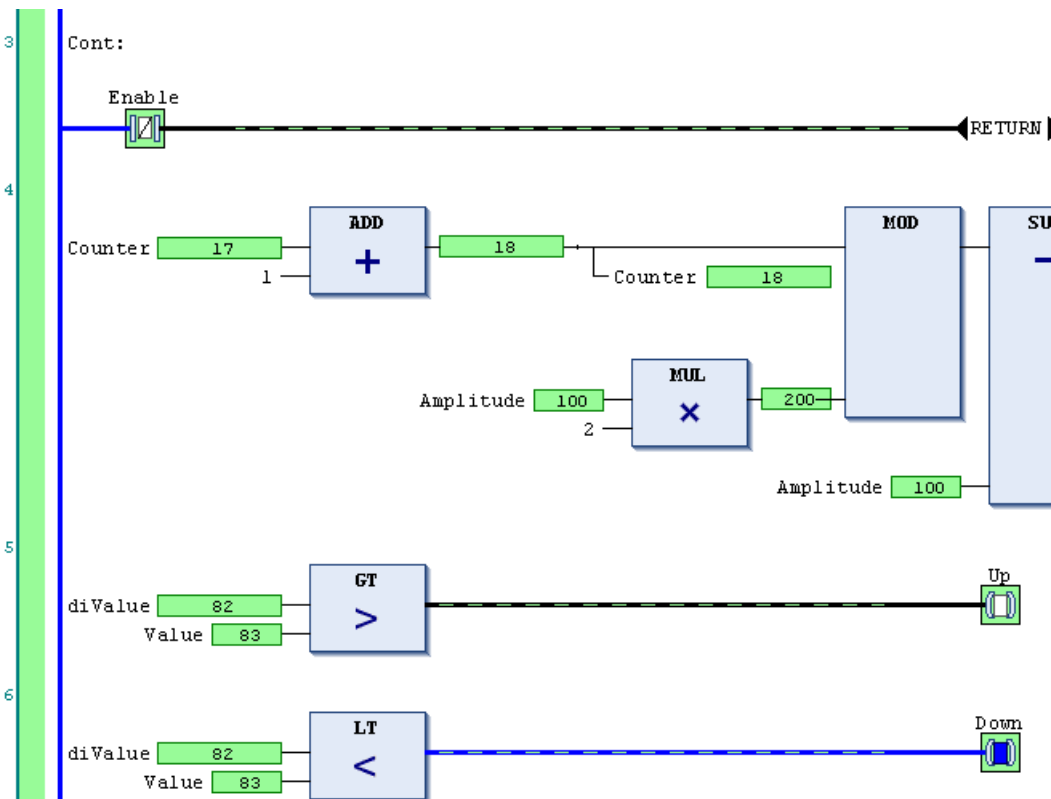
```

1  i 1619 := i 1619 + 1;
2  b 0 := NOT b 0;
3  IF str 'abcdefghij' = str1 "" THEN
4  f12 1.5 := f1 1.23;
5  ELSE
6  f12 1.5 := 1.5;
7  D 6.5E+04 := B255 * B255;
8  END_IF;
9  IF D 6.5E+04 < 0.0 THEN

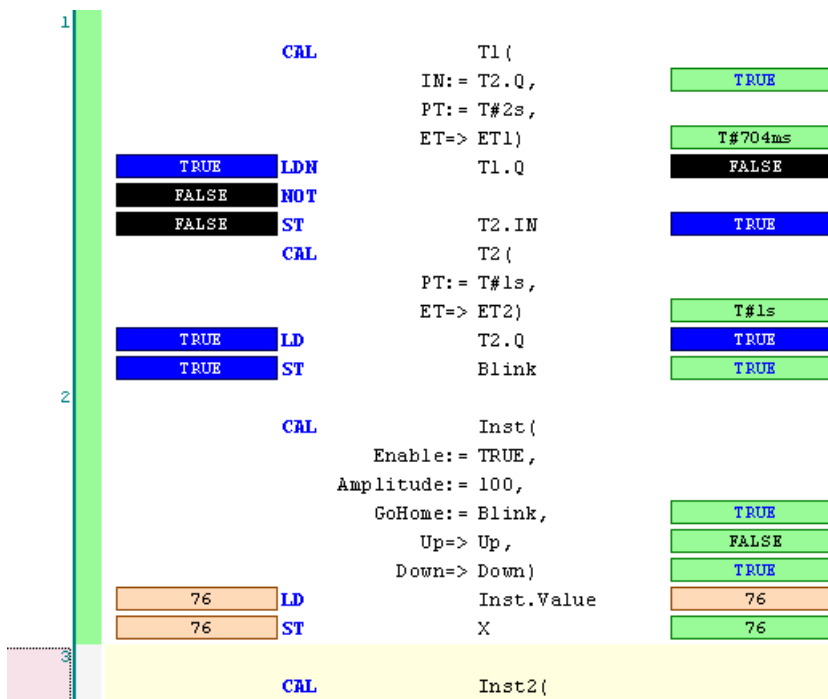
```

In network editors, TwinCAT marks the networks that have been run through by bars at the left-hand edge in the "flow control color".

In LD, TwinCAT displays the connecting lines currently being run through in green, the others in gray. The actual value of the connection is also displayed: TRUE by thick blue lines, FALSE by thick black lines, unknown or analog values by thin black lines. This can lead in the case of combination of the respective pieces of information to dashed lines.



In IL, TwinCAT uses two fields for each statement for displaying the actual values. One to the left of the operator with the current accumulator value and one to the right of the operand with the operand value.



See also:

- TC3 User Interface documentation: [Command Flow Control \[▶ 960\]](#)

9.6 Determining the Current Processing Position with the Call Stack

With the aid of the call stack you can determine the current position of the program execution. This function is very useful for the step-by-step execution of the program.

- ✓ The project is in online mode. The program has stopped at a breakpoint or you are executing it step by step.
1. Open the call stack with the command **Call Stack** in the menu **PLC > Windows**.
 - ⇒ The call stack is opened. The list shows the currently reached position with the full call path.

The call stack is also available in offline mode or in normal online mode (without currently using the debugging functions). In this case it contains the position last displayed during a step-by-step execution in "grayed" lettering.

See also:

- [Stepwise processing of the program \(stepping\) \[▶ 213\]](#)
- TC3 User Interface documentation: [Command Call Stack \[▶ 945\]](#)

10 PLC project at runtime

When the application program is running on the PLC there are functions in the TwinCAT 3 development system for monitoring and changing the values of variables as well as the recording and saving of their development.

10.1 Monitoring Values

At runtime you can observe the current values of variables of a programming object at various points in the project. This is called "monitoring":

- Online view of the programming editor of an object: "inline monitoring"
- Online view of the declaration editor of an object
- Object-independent configurable watch lists

You can monitor the results of function calls and the current values of variables in objects of the type **Property** by setting the pragma {attribute 'monitoring'...}.


See also:

- Reference programming: Pragmas > 'monitoring' attribute [▶ 810]

10.1.1 Monitoring in Programming Objects

- ✓ You have loaded a project into the controller and started it.
You wish to observe the current values that the variables on the controller adopt in the TwinCAT development system in the editors of the programming blocks.
 - 1. Make sure that the option **Enable inline monitoring** is activated in the menu **Tools > Options** in the category **TwinCAT > PLC Environment > Text editor** on the tab **Monitoring**.
 - 2. Select the command **Decimal** in the menu **PLC > Display Mode** to set the display format of the values.
 - 3. Double-click on the POU in the PLC project tree in the Solution Explorer to open the associated editor.
- ⇒ You can see the representations of the variable values in the declaration part and implementation part. And you will find a general description of this plus the special features that depend on the type of POU and the programming editor.

Monitoring in the declaration editor

The current value of a variable is displayed in the column **Value**. You can write and force the value entered in the column **Prepared value**. Forced values are preceded by a red symbol .

TwinCAT_Device.Project3.MAIN					
Expression	Type	Value	Prepared value	Address	Comment
nVar2	INT	F 2411		%IB10	
bVar3	BOOL	FALSE	TRUE	%IX0.0	
aVar	ARRAY [0..3] O...				
aVar[0]	INT	0			
aVar[1]	INT	0			
aVar[2]	INT	0			
aVar[3]	INT	0			
stMyStruct	ST_MyStruct				
nTest1	INT	0			
nTest2	INT	0			
fbSample	FB_Sample				instance of function block FB_Sample
nIn	INT	0			
nOut	INT	0			
nVar1	INT	0			
nRes	INT	0			

The printout of an interface reference is expandable. If the interface points to a global instance, this global instance is displayed as the first entry under the reference. If the interface reference changes afterwards, the displayed reference collapses.

See also:

- [Declaration editor \[► 622\]](#)
- [Forcing and writing variables \[► 214\]](#)

Monitoring in implementation (inline monitoring)

The display of the current variable value in the implementation part is called inline monitoring. The following displays are possible, depending on the programming language:

- Variables have a window with the current value displayed behind their name: `nResult` 17
If you have prepared values for forcing or writing for variables, these are displayed within the inline monitoring window behind the current value in angle brackets.
After forcing, the affected values are marked with the symbol **F**
- Network editors and CFC:
Connecting lines (signal lines) are color-coded according to their Boolean actual value: blue means TRUE, black means FALSE.
- LD editor:
In addition, the contact and coil elements are marked. For contacts or coils, a prepared value (TRUE or FALSE) is displayed in a small window next to the element.
- SFC editor:
Transitions with value TRUE are color-coded according to their Boolean actual value: blue means TRUE, black means FALSE.
Active steps are marked in blue.
In the implementation forced transition values are marked red.
- IL editor:
Current values are displayed in a separate column

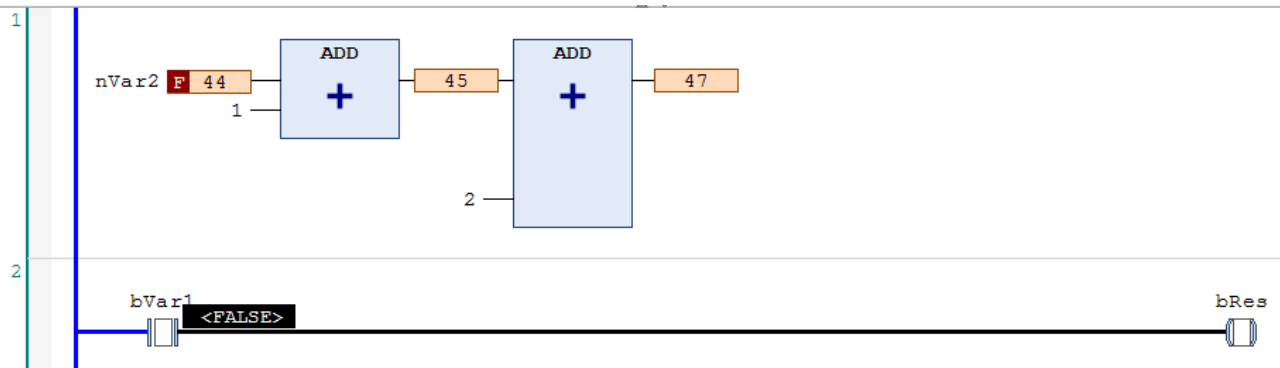
Monitoring in the ST editor:

```

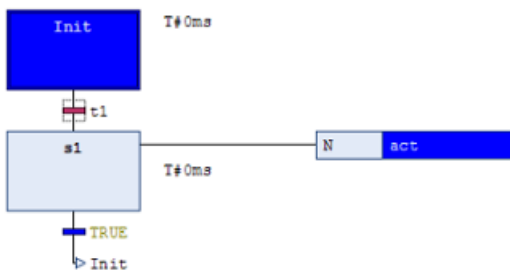
1 nVar2(44) := nVar2(44) + 1; (*counter*)
2 bVar3(TRUE) := TRUE;
3 stMyStruct.nTest1(45) := nVar2(44);
4 aVar[2](45) := nVar2(44);
5 nRes(0) := fbSample.nOut(0);
6

```

Monitoring in the LD editor:



Monitoring in the SFC editor:



You can deactivate the inline monitoring function here: Menu **Tools > Options**, Category **TwinCAT > PLC Environment > Text editor**, Tab **Monitoring**.

See also:

- Reference Programming: [ST editor in online mode](#) [► 625]
- Reference programming: [FBD/LD/IL editor in online mode](#) [► 655]
- Reference programming: [SFC editor in online mode](#) [► 635]
- Reference programming: [CFC editor in online mode](#) [► 672]

Partially monitor array

An array that is expanded shows the actual values for up to 1000 elements. This can be confusing. Moreover, an array can contain more than 1000 elements. Then it is helpful to limit the range of displayed elements. You can do this during online operation in the following ways.

- ✓ You have loaded a project on the controller in which a multi-dimensional array variable with more than 1000 elements is declared.

Sample: `aSample : ARRAY [1..100, 1..10, 1..20] OF INT;`

1. For the variable `aSample`, click in the field of the column **Data type**.

⇒ The **Monitoring area** dialog opens.

2. Enter the value [1, 0, 0] at **Start**.

3. Enter the value [1, 10, 20] at **End**.

⇒ The actual values of 200 array elements are displayed. The range is limited to elements of index [1, <i>, <j>].

Monitoring a function block

If you are about to open an editor view for a function block in online mode, you will be asked whether you wish to open the view of the basic implementation or an instance of the function block. Monitoring is only possible in the basic implementation if a breakpoint is set. If you set a breakpoint in the basic implementation, it will be set in all instances. The values of the instance that is executed first in the program sequence are displayed first in the basic implementation view.

If you double-click on the editor view of a function block during online operation, a dialog appears in which you can choose between the view of the basic implementation or a specific instance.


If you select the basic implementation, then the code appears in the editor without displaying current values. Now set a breakpoint in the basic implementation. If the execution stops there, the current values of that instance are displayed which is processed first in the program flow. You can now step successively through all instances.

If you select one of the instances, then the editor with the code of the function block instance opens. The current values are displayed and constantly updated in the declaration and, if necessary, in the implementation.

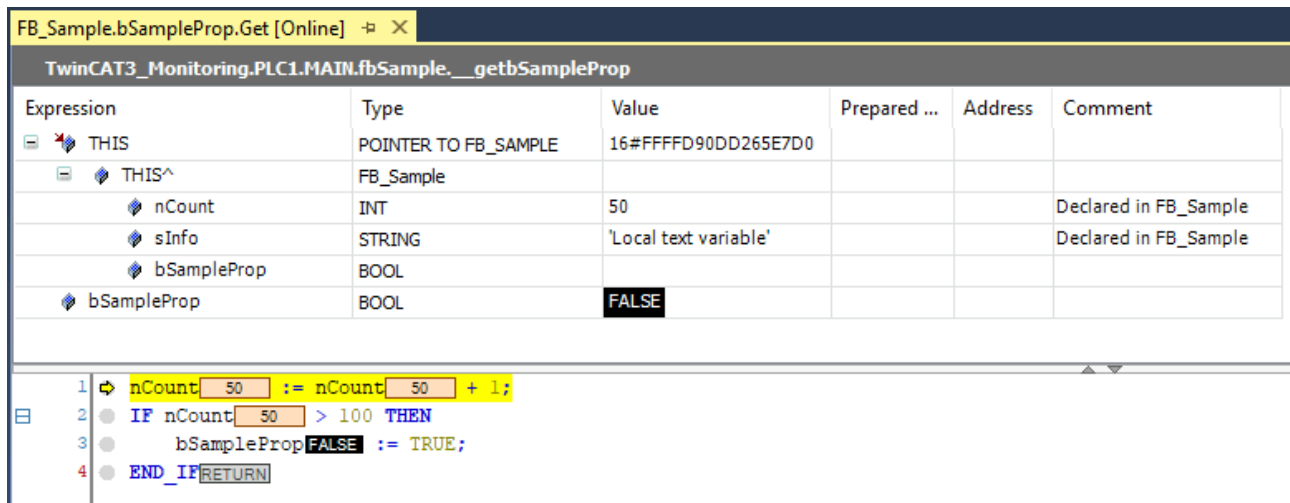
See also:

- [Object Function block \[► 84\]](#)
- [Use of breakpoints \[► 211\]](#)

Monitoring the property

You can monitor variables in a property object  when you set a breakpoint during online operation. When stopped there, the current values are displayed.

In addition to its own values, the values of the variables of the higher-level instance are automatically displayed. In the declaration part of the property, the pointer THIS appears in the first line, pointing to the higher-level instance, with the data type specifications and current values.



Expression	Type	Value	Prepared ...	Address	Comment
THIS	POINTER TO FB_SAMPLE	16#FFFFD90DD265E7D0			
THIS^	FB_Sample				
nCount	INT	50			Declared in FB_Sample
sInfo	STRING	'Local text variable'			Declared in FB_Sample
bSampleProp	BOOL				
bSampleProp	BOOL	FALSE			

```

1 nCount 50 := nCount 50 + 1;
2 IF nCount 50 > 100 THEN
3     bSampleProp FALSE := TRUE;
4 END_IF RETURN
    
```

See also:

- [Object Property \[► 96\]](#)

Monitoring the property access in the higher-level programming object


You can monitor the values of lower-level properties  in a function block or program in addition to the variable values.

To do this, insert either the pragma {attribute 'monitoring' = 'variable'} or the pragma {attribute 'monitoring' = 'call'} to the lower-level property object in the declaration. If you open the higher-level program or the higher-level function block instance at runtime, then the current property values are displayed in the editor in addition to the current variable values.

See also:

- Reference programming: Pragmas > 'monitoring' attribute [► 810]

Monitoring the method

You can monitor variables in a method object  if you set a breakpoint in the method during online operation. When stopped there, the current values are displayed.

In addition to its own values, the values of the variables of the higher-level instance are automatically displayed. In the declaration part of the method, the pointer THIS appears in the first line for this purpose, which points to the higher-level instance, with the data type specifications and current values.

FB_Sample.SampleMethod [Online] -> X					
TwinCAT3_Monitoring.PLC1.MAIN.fbSample.SampleMethod					
Expression	Type	Value	Prepared...	Address	Comment
THIS	POINTER TO FB_SAMPLE	16#FFFFFF90DD265E7D0			
THIS^	FB_Sample				
nCount	INT	51			Declared in FB_Sample
sInfo	STRING	'Local text variable'			Declared in FB_Sample
bSampleProp	BOOL				
SampleMethod	BOOL	FALSE			
nCountMethod	INT	0			


```

1 | nCountMethod[ 0 ] := nCountMethod[ 0 ] + 1; RETURN

```

See also:

- Object Method [► 90]

Monitoring the function

You can monitor variables in a function object if you set a breakpoint in the function during online operation. When stopped there, the current values are displayed.

- Object Function [► 81]

Monitoring the return value of a function call

In the ST editor, the current return value of a function is displayed in the inline monitoring instead of a POU at which the function is called if the following is satisfied:

- it is a value that can be interpreted as a 4-byte numerical value (e.g. INT, SINT or LINT).
- the pragma {attribute 'monitoring' := 'call'} is inserted in the function declaration.

See also:

- Reference Programming: Pragmas > Attribute 'monitoring' [► 810]




10.1.2 Using Watchlists

What is a watch list?

A watch list is a user-defined list of project variables that are collected in a view for the purpose of monitoring their values. You can write and force variable values in a watch list in online mode.

Four watch lists **Watch <n>** are available for filling in the menu **PLC > Windows** in a project.

Creating and editing a watch list (offline or online mode)

- ✓ A project is opened in offline or online mode. Variables are declared in the project that you wish to have in one of the four watch lists.
 - 1. Select the command **Watch <n>** in the menu **PLC > Windows**.
 - ⇒ The view **Watch <n>** appears. It contains an empty table row.
 - 2. After double-clicking on the field in the column **Expression**, enter a variable to be monitored either manually or using the **Input Assistant**.
 - Syntax: <Device name>.<Project name>.<Object name>.<Variable name>
 - Example: „TwinCAT_Device.Project5.MAIN.nVar“
 - If you enter the name of a structured variable, the individual components will be displayed automatically in further rows in online mode.
 - 3. One after the other, define all the variables to be monitored with this list. You can change the order with drag-and-drop.
- ⇒ The fields **Execution point**, **Type**, **Address**, **Comment** and **Value** are populated automatically according to the variable declaration. The symbol in front of the expression indicates whether the variable is  an input variable,  an output variable or  a "normal" variable.



In online mode you can also create a new watch list or edit an existing one with the aid of the context menu command **Add Watch**.

See also:

- TC3 User Interface documentation: [Command Watch List <n> \[► 939\]](#)

Addition of variables with the help of the command **Add Watch** (online mode)

- ✓ A project is open in operation. Variables that you wish to place in a watch list are already declared in the project.
- 1. Open the desired watch list with the command **Watch <n>** in the menu **PLC > Windows**.
- 2. Place the cursor on the variable in the declaration or implementation part of a POU and select the command **Add Watch** from the context menu.
 - ⇒ An entry for the selected variable is added to the list.
- 3. You can add further variables in this way or by entering them in the list in the field **Expression**, as described above (**Creating and editing a watch list**).
 - ⇒ The watch lists are updated immediately.



If no watch list is open when you apply the command **Add Watch** to a variable, this is added automatically to the list "Watch 1".



The writing and forcing of variable values is also possible in the watch lists. The column **Prepared value** appears for this in online mode.

See also:

- [Forcing and Writing Variables Values \[► 214\]](#)
- TC3 User Interface documentation: [Command Watch List <n> \[► 939\]](#)
- TC3 User Interface documentation: [Command Add to Watch \[► 879\]](#)

10.2 Changing Values with Recipes

You use recipes to change or read out the values for a certain set of variables (recipe definitions) on the controller at the same time.

The basic settings for the recipes such as storage location and format are defined in the object **Recipe Manager**. Below this object you add one or more recipe definitions. A **Recipe Definition** encompasses one or more recipes for the variables contained in it. The recipe consists of certain variable values.


You can save the recipes in files or write them directly from files into the controller.

Recipes can be loaded via the TwinCAT programming interface, via visualization elements or via the application program.



For the use of recipes in your application program it is recommended to call this program section via its own task with low priority.

10.2.1 Object Recipe Manager

Symbol: 

The recipe manager offers functions for the administration of user-defined variable lists: so-called recipe definitions. The recipe definitions can be stored in "Recipe files".

If you add a recipe manager to the PLC project, the library RecipeManagement is automatically added in the library manager.

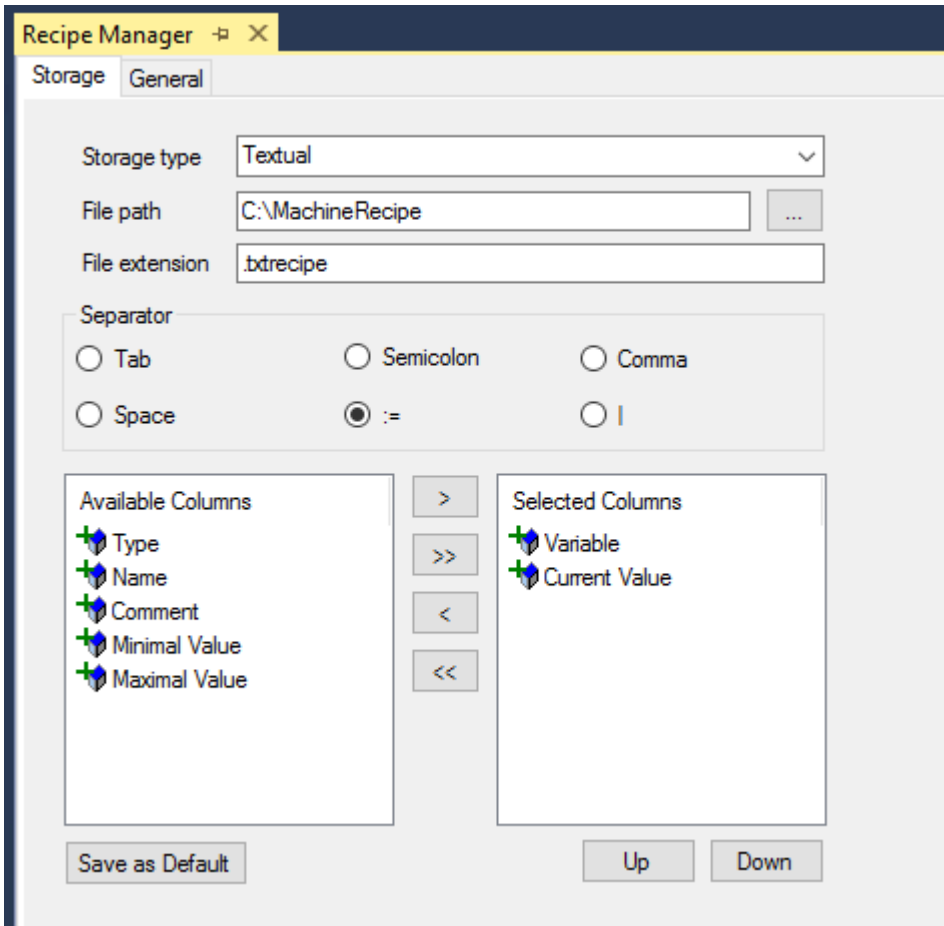
Create object Recipe Manager

1. Select the PLC project in the **Solution Explorer**.
2. Select the command **Add > Recipe Manager...** in the context menu.
 - ⇒ The dialog **Add Recipe Manager** opens.
3. Enter a name.
4. Click on **Open**.
 - ⇒ The recipe manager is added to the PLC project tree and opened in the editor. Subsequently, you can add the object **Recipe Definition** to the object **Recipe Manager**.

Structure of the recipe manager

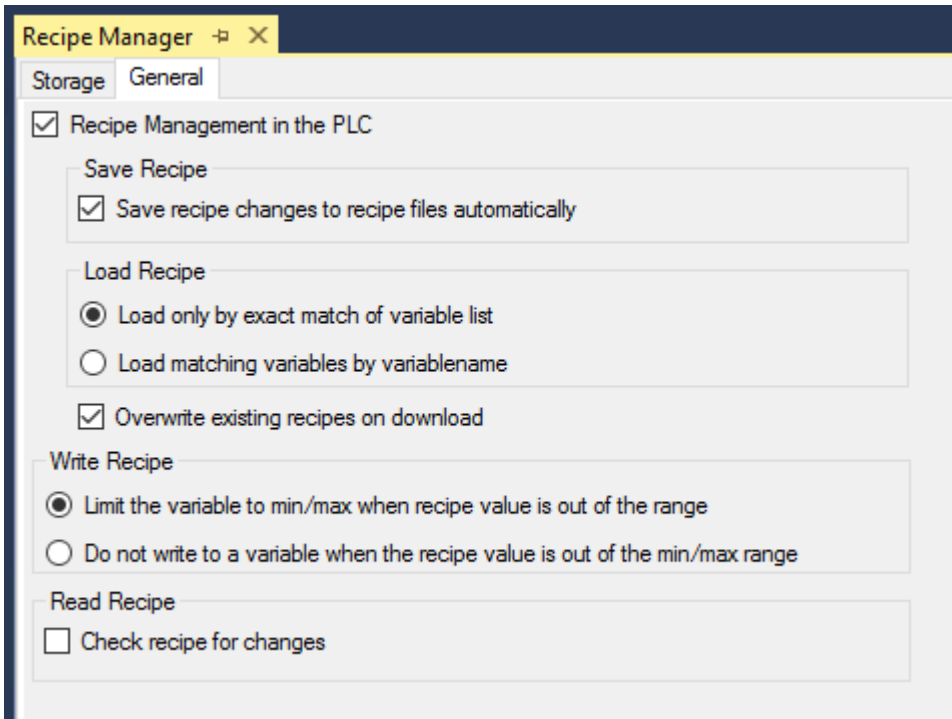
The editor of the recipe manager is comprised of the two tabs **Storage** and **General**, in which the settings for the recipe manager are determined.

Tab Storage



Storage Type	<p>textual: TwinCAT saves the recipe in a readable format with the configured columns and separators.</p> <p>binary: TwinCAT saves the recipe in a non-readable binary format. This format requires less storage space.</p> <p>You can only read recipes saved in binary format back in if you have not changed the variable lists.</p>
File path	<p>Relative path on the runtime system.</p> <p>This path is created on the target system in the directory of the runtime files. TwinCAT creates a file for each recipe in this directory on downloading to the controller. The prerequisite is that the option Recipe Management in the PLC is activated.</p> <p>The files are loaded into the recipe manager each time the PLC project is restarted.</p>
File extension	<p>File extension for the recipe file.</p> <p>This results in the standard name for recipe files in the form <Recipe>.<Recipe definition>.<File extension>.</p>
Separator	<p>Separators between the individual values in the stored file.</p>
Available Columns Selected Columns	<p>Definition stating what information is saved in the recipe file and in what order.</p>
Save as default	<p>TwinCAT uses the settings in the dialog immediately as default settings.</p>

Tab General



Recipe Management in the PLC	<input checked="" type="checkbox"/> : Must be activated if recipes are loaded at runtime via visualization elements or via the user program. The option can be deactivated if recipes are transferred to the controller exclusively via the TwinCAT programming interface.
------------------------------	--

Save Recipe

Save recipe changes to recipe files automatically	<input checked="" type="checkbox"/> : Recommended option, because it causes the "usual" behavior of a recipe management. At runtime the recipe manager saves the recipes automatically to a file in the case of a change; i.e. the memory files are updated automatically on each change of a recipe at runtime. The option is only effective if the option Recipe management in the PLC is activated.
---	---

Load Recipe

Load only by exact match of variable list	The recipe is only loaded if the file contains all variables from the variable list of the recipe definition of the PLC project and these are sorted in the same order. Additional entries at the end are ignored. If the necessary match is not present, the error state ERR_RECIPE_MISMATCH is set (RecipeManCommands.GetLastError).
Load matching variables by variablename	The recipe values are only loaded for those variables that have the same name in the recipe definition of the PLC project as in the recipe file. If the variable lists differ in composition and sorting, no error state is set. This allows recipe files to be loaded even if variables have been deleted from the file or the recipe definition.
Overwrite existing recipes during download	<input checked="" type="checkbox"/> : If recipe files with the same name exist on the controller, they are overwritten with the configured values from the project when the PLC project is started. If the values are to be loaded from the existing recipe files instead, this option must be deactivated. Prerequisite: Storage type is textual and the option Save recipe changes to recipe files automatically is activated.

Write Recipe

Limit variable to min/max if recipe value out of range	If the recipe contains a value that is outside the value range entered in the definition, the defined minimum or maximum value is written to the PLC variable instead of this value.
Do not write variable if recipe value outside min/max range	If the recipe contains a value that is outside the value range entered in the definition, no value is written to the PLC variable. It retains its current value.

Read Recipe

Load only by exact match of variable list	<p><input checked="" type="checkbox"/> : With each method call, the current PLC variable values are first read into the recipe. Then it is checked whether the values have changed. Only if the values have changed, the recipe is saved, i.e. the recipe file is overwritten with the current recipes.</p> <p>The option can be used to update the recipe file located in the local file system only if recipe values have changed on the PLC. However, this affects performance because additional code is generated for the check.</p> <p><input type="checkbox"/> : With each method call, the current PLC variable values are first read into the recipe. Then the recipe is written to the recipe file in the local file system.</p> <p>Since each call writes to the file system, the controller can be loaded.</p>
---	--


Recipes in online mode, if the option **Save recipe changes to recipe files automatically** is activated:

Actions	Recipes defined in the project	Recipes defined at runtime
Online reset warm Online reset cold Download	The recipes of all recipe definitions are assigned the values from the current project.	Dynamically generated recipes remain unchanged.
Online reset origin	The application is removed from the PLC. If this is followed by a new download, the recipes will be restored as in the case of an online reset warm.	
Shutdown and restart of the PLC	After the restart the recipes are reloaded from the automatically created files. The same state as before shutting down is thus restored.	
Online Change	The recipe values remain unchanged. During runtime a recipe can only be changed using the commands from the function block RecipeManCommands.	
Stop/start	The recipes remain unchanged in the case of a stop/start of the PLC.	

Recipes in online mode, if the option **Save recipe changes to recipe files automatically** is NOT activated:

Actions	Recipes defined in the project	Recipes defined at runtime
Online reset warm Online reset cold Download	The recipes of all recipe definitions are assigned the values from the current project. These are only set in the memory, however. The Save recipe command must be used explicitly to store the recipes in a file.	Dynamically generated recipes will be lost.
Online reset origin	The application is removed from the PLC. The recipes will be restored if this is followed by a new download.	Dynamically generated recipes will be lost.
Shutdown and restart of the PLC	After the restart the recipes are reloaded from the automatically created files. The same state as before shutting down is thus restored.	
Online Change	The recipe values remain unchanged. During runtime a recipe can only be changed using the commands from the function block RecipeManCommands.	
Stop/start	The recipes remain unchanged in the case of a stop/start of the PLC.	

10.2.2 Object Recipe Definition

Symbol: 

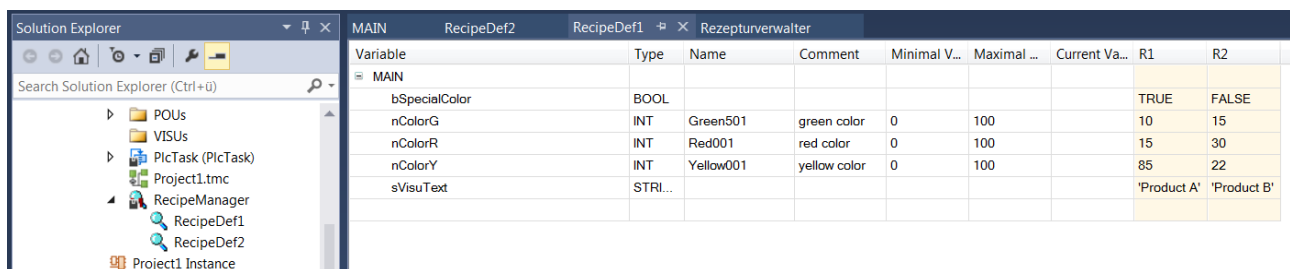
Within a recipe definition you define various value sets for the variables, which are called recipes.

Create object Recipe Definition

1. Select the object **Recipe Manager** in the **Solution Explorer** in the PLC project tree.
2. Select the command **Add > Recipe Definition...** in the context menu.
 - ⇒ The dialog **Add Recipe Definition** opens.
3. Enter a name.
4. Click on **Open**.
 - ⇒ The recipe definition is added underneath the recipe manager and opened in the editor.

Structure of a recipe definition

The value sets are displayed in tabular form in the recipe definition editor. You can switch the view of the recipe definition between the simple view and the structured view. TwinCAT displays the variables belonging to a structure grouped together in the structured view.



Variable	Type	Name	Comment	Minimal V...	Maximal ...	Current Va...	R1	R2
MAIN								
bSpecialColor	BOOL						TRUE	FALSE
nColorG	INT	Green501	green color	0	100		10	15
nColorR	INT	Red001	red color	0	100		15	30
nColorY	INT	Yellow001	yellow color	0	100		85	22
sVisuText	STRI...						'Product A'	'Product B'

Variable	Name of the variable
Type	Entered automatically
Name	Optional
Minimum value Maximum value	If the variable value is smaller than the minimum value or larger than the maximum value, TwinCAT sets the value to the minimum or maximum value respectively.
Comment	Additional information, for example the unit of the value.
Current value	Current variable value; displayed in online mode only.
Recipe (e.g. R1)	Variable values of the recipe

10.2.3 Using recipes

Handling recipes in the TwinCAT user interface

The TwinCAT programming interface provides you with commands for the creation of recipes as well as reading/writing in online mode.

See also:

- TC3 User Interface documentation: [Recipes \[► 1049\]](#)

Use of recipes in the application

You can use recipes at runtime in the user program or via visualization elements.

In the user program you use methods of the function block `RecipeManCommands` from the library `RecipeManagement`. In the visualization the use of recipes takes place via the input configuration (internal command) of visualization elements.




During the initialization procedure the recipe management reads the values of variables that are defined in the recipe definition. This procedure takes place at the end of the initialization phase of the application. All initial values of the application variables are set at this point in time. This is carried out so that missing values from recipe files can be correctly initialized.

See also:

- [Library Recipe Management - RecipeManCommands \[► 235\]](#)

Creating a recipe

1. Select the PLC project in the **Solution Explorer**.
2. Select the command **Add > Recipe Manager...** in the context menu.
 - ⇒ TwinCAT adds the recipe manager to the PLC project.
3. Select the object **Recipe Manager** in the PLC project tree.
4. Select the command **Add > Recipe Definition...** in the context menu.
 - ⇒ TwinCAT adds the recipe definition below the recipe manager.
5. Open the recipe definition editor by double-clicking on the object.
6. Double-click on the empty field in the column **Variable** in the editor. Enter the name of a variable for which you wish to define a recipe. The input assistant is available for this (button ).
7. Select the command **Add Recipe** in the menu **Recipe** or in the context menu of the editor and enter a name for the new recipe (e.g. "MyRec").
 - ⇒ A column with the recipe name appears in the editor.
8. Enter the value of the variable for this recipe.
9. Add further variables if necessary.
10. Select a variable value of the recipe and execute the command **Save Recipe** in the menu **Recipe** or the context menu. Select a memory location and a file name.

⇒ TwinCAT saves the recipe in the format defined in the recipe manager.

● **Overwriting the implicit recipe file**

I The implicitly used recipe files, which are needed as a clipboard for reading and writing recipes, must not be overwritten. This means that the file name must be different to *<Recipe name>.<Recipe definition name>.txtrecipe*

See also:

- TC3 User Interface documentation: [Command Add a new recipe \[► 1049\]](#)
- TC3 User Interface documentation: [Command Save recipe \[► 1051\]](#)

Loading a recipe from a file

- ✓ There is a recipe management in a PLC project. There is a recipe "MyRec" with variable values in a recipe definition. A recipe file *MyRec.txt* containing the entries for this recipe exists in the file system.
- 1. Open the table editor for the definition of the individual recipes by double-clicking on the object **Recipe Definition** in the PLC project tree.
 - ⇒ You will see a column **MyRec** containing the current values for this recipe.
- 2. Edit the file *MyRec.txt* in an external text editor and replace the variable values by others that you wish to load into the recipe definition in TwinCAT. Save the file.
- 3. Click in the column **MyRec** in the recipe definition and select the command **Load Recipe** in the menu **Recipe** or in the context menu.
 - ⇒ The dialog **Load Recipe** opens.
- 4. Select the file *MyRec.txt* from the File Explorer for loading.
 - ⇒ The recipe values in the recipe definition are updated according to the values read in the file. If you change the current values of the recipe variables by loading the recipe, a query appears the next time you log in asking whether you wish to log in with online change, download or without changes.

Example of a recipe file:

```
MAIN.nVar1:=0
MAIN.nVar2:=2
MAIN.nVar3:=35232
MAIN.sVar4:='first'
MAIN.wsVar5:='123443245'
```

I If you want to overwrite only individual recipe variables with new values, remove the values for the other variables before loading the recipe into the recipe file. Entries without value specification are not read, which means that these variables are unaffected by the update on the controller and in the project.

For values of the REAL/LREAL data type, the hexadecimal value is also written to the recipe file in some cases. This is necessary so that the exact identical value is restored during the back conversion process. In this case, change the decimal value and delete the hexadecimal value.

See also:

- TC3 User Interface documentation: [Command Load Recipe \[► 1050\]](#)

Reading a recipe

- ✓ TwinCAT is in online mode.
- 1. Click in the recipe column in the recipe definition and select the command **Read Recipe** in the menu **Recipe** or in the context menu.
 - ⇒ TwinCAT overwrites the values of the selected recipe with the values read from the controller. In the process, the values are implicitly stored (in a file on the controller) and simultaneously displayed in the table of the recipe definition.

See also:

- TC3 User Interface documentation: [Command Read Recipe \[► 1051\]](#)
- TC3 User Interface documentation: [Command Read and Save Recipe \[► 1052\]](#)

Writing a recipe

- ✓ TwinCAT is in online mode.
- 1. Click in the recipe column in the recipe definition and select the command **Write Recipe** in the menu **Recipe** or in the context menu.
- ⇒ TwinCAT overwrites the values in the controller with the values from the selected recipe.

See also:

- TC3 User Interface documentation: [Command Write Recipe \[▶ 1051\]](#)
- TC3 User Interface documentation: [Command Load and Write Recipe \[▶ 1052\]](#)

10.2.4 Library Recipe Management - RecipeManCommands

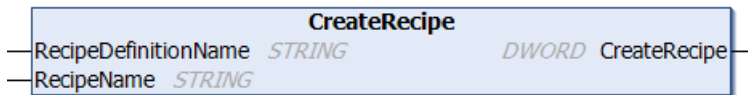
The methods of the function block RecipeManCommands from the library "Recipe Management" enable the recipes to be managed programmatically.

i The application automatically creates recipe files with the name *<Recipe>.<Recipe definition>.txtrecipe* on the controller. They serve as a clipboard for reading and writing recipe variables. The option **Save recipe changes to recipe files automatically** on the tab **Recipe Manager > General** influences access to these files.

i If the option **Save recipe changes to recipe files automatically** is activated, the recipes of the definition in TwinCAT and the implicit recipe files on the controller are automatically kept identical. The changing of recipes then also leads to file accesses.

Method CreateRecipe

This method creates a new recipe in the specified recipe definition. It subsequently reads the current PLC values into the new recipe and saves it as a recipe file with a standard name. The standard name is *<Recipe>.<Recipe definition>.<Recipe extension>*.



Inputs (VAR_INPUT)

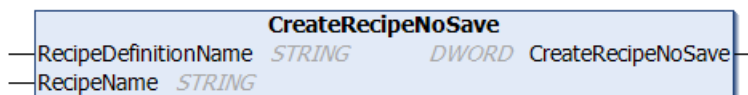
Name	Data type	Description
RecipeDefinitionName	STRING	Name of the recipe definition
RecipeName	STRING	Name of the recipe

Return value

Name	Data type	Description
CreateRecipe	DWORD	ERR_RECIPE_DEFINITION_NOT_FOUND ERR_RECIPE_ALREADY_EXIST ERR_RECIPE_NOMEMORY ERR_NO_RECIPE_MANAGER_SET ERR_OK

Method CreateRecipeNoSave

This method creates a new recipe in the specified recipe definition. It subsequently reads the actual values into the new recipe.



Inputs (VAR_INPUT)

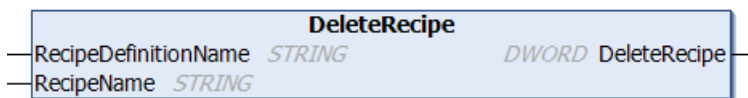
Name	Data type	Description
RecipeDefinitionName	STRING	Name of the recipe definition
RecipeName	STRING	Name of the recipe

Return value

Name	Data type	Description
CreateRecipeNoSave	DWORD	ERR_RECIPE_DEFINITION_NOT_FOUND ERR_RECIPE_ALREADY_EXIST ERR_RECIPE_NOMEMORY ERR_NO_RECIPE_MANAGER_SET ERR_OK

Method DeleteRecipe

This method deletes a recipe from the recipe definition.



Inputs (VAR_INPUT)

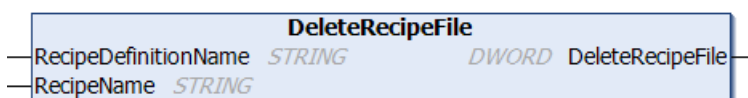
Name	Data type	Description
RecipeDefinitionName	STRING	Name of the recipe definition
RecipeName	STRING	Name of the recipe

Return value

Name	Data type	Description
DeleteRecipe	DWORD	ERR_RECIPE_DEFINITION_NOT_FOUND ERR_RECIPE_NOT_FOUND ERR_NO_RECIPE_MANAGER_SET ERR_OK

Method DeleteRecipeFile

This method deletes the specified recipe file of a recipe. The recipe file must be stored under the standard name <Recipe>.<Recipe definition>.<Recipe extension>.



Inputs (VAR_INPUT)

Name	Data type	Description
RecipeDefinitionName	STRING	Name of the recipe definition
RecipeName	STRING	Name of the recipe

Return value

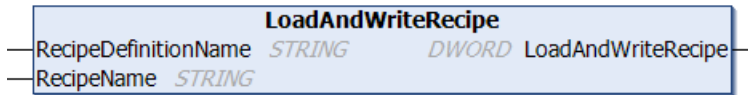
Name	Data type	Description
DeleteRecipeFile	DWORD	ERR_RECIPE_DEFINITION_NOT_FOUND ERR_RECIPE_NOT_FOUND ERR_RECIPE_FILE_NOT_FOUND ERR_NO_RECIPE_MANAGER_SET ERR_OK

Method LoadAndWriteRecipe

The method loads a recipe from the specified recipe file. The recipe file must be stored under the standard name <Recipe>.<Recipe definition>.<Recipe extension>. It subsequently writes the recipe into the PLC variables.



Entries in the recipe file that contain no value assignment are not loaded and written. Refer to the description of the menu command **Load and write Recipe**.



Inputs (VAR_INPUT)

Name	Data type	Description
RecipeDefinitionName	STRING	Name of the recipe definition
	STRING	
RecipeName	STRING	Name of the recipe

Return value

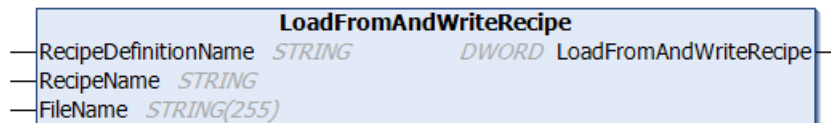
Name	Data type	Description
LoadAndWriteRecipe	DWORD	ERR_RECIPE_DEFINITION_NOT_FOUND ERR_RECIPE_NOT_FOUND ERR_RECIPE_FILE_NOT_FOUND ERR_RECIPE_MISMATCH ERR_RECIPE_NOT_ALL_VARIABLES_WERE_LOADED ERR_NO_RECIPE_MANAGER_SET ERR_OK

Method LoadFromAndWriteRecipe

This method loads the specified recipe file into a recipe. It subsequently writes the recipe into the PLC variables.



Entries in the recipe file that contain no value assignment are not loaded and written. Refer to the description of the menu command **Load and write Recipe**.



Inputs (VAR_INPUT)

Name	Data type	Description
RecipeDefinitionName	STRING	Name of the recipe definition
RecipeName	STRING	Name of the recipe
FileName	STRING (255)	Name of the file

Return value

Name	Data type	Description
LoadFromAndWriteRecipe	DWORD	ERR_RECIPE_DEFINITION_NOT_FOUND ERR_RECIPE_NOT_FOUND ERR_RECIPE_FILE_NOT_FOUND ERR_RECIPE_MISMATCH ERR_RECIPE_NOT_ALL_VARIABLES_WERE_LOADED ERR_NO_RECIPE_MANAGER_SET ERR_OK

See also:

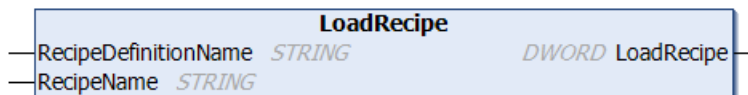
- TC3 User Interface documentation: [Command Load and Write Recipe \[► 1052\]](#)

Method LoadRecipe

The method loads a recipe from a recipe file. The recipe file must be stored under the standard name <Recipe>.<Recipe definition>.<Recipe extension>.



Entries in the recipe file that contain no value assignment are not loaded and written. Refer to the description of the menu command **Load and write Recipe**.



Inputs (VAR_INPUT)

Name	Data type	Description
RecipeDefinitionName	STRING	Name of the recipe definition
RecipeName	STRING	Name of the recipe

Return value

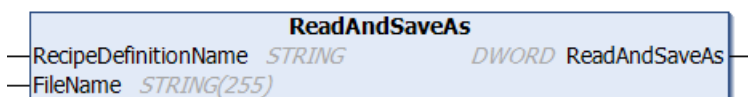
Name	Data type	Description
LoadRecipe	DWORD	ERR_RECIPE_DEFINITION_NOT_FOUND ERR_RECIPE_NOT_FOUND ERR_RECIPE_FILE_NOT_FOUND ERR_RECIPE_MISMATCH ERR_RECIPE_NOT_ALL_VARIABLES_WERE_LOADED ERR_NO_RECIPE_MANAGER_SET ERR_OK

See also:

- TC3 User Interface documentation: [Command Load and Write Recipe \[► 1052\]](#)

Method ReadAndSaveAs

This method reads the current PLC values from the variables of the recipe definition and saves this data set in a recipe file without changing the existing standard recipe file <recipe.recipedefinition.extension>.



Inputs (VAR_INPUT)

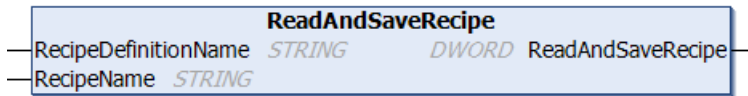
Name	Data type	Description
RecipeDefinitionName	STRING	Name of the recipe definition The variables specified in the recipe definition are read out.
FileName	STRING(255)	Name of the file. The currently read-out data set is saved as a recipe in the file.

Return value

Name	Data type	Description
ReadAndSaveAs	DWORD	ERR_RECIPE_DEFINITION_NOT_FOUND ERR_RECIPE_SAVE_ERR ERR_NO_RECIPE_MANAGER_SET ERR_OK

Method ReadAndSaveRecipe

This method reads the current PLC values into the recipe. It subsequently saves the recipe to a recipe file with a standard name. The standard name is <Recipe>.<Recipe definition>.<Recipe extension>. The contents of any existing file are overwritten.



Inputs (VAR_INPUT)

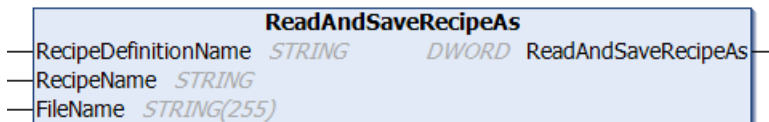
Name	Data type	Description
RecipeDefinitionName	STRING	Name of the recipe definition
RecipeName	STRING	Name of the recipe

Return value

Name	Data type	Description
ReadAndSaveRecipe	DWORD	Return value, possible values: ERR_RECIPE_DEFINITION_NOT_FOUND ERR_RECIPE_NOT_FOUND ERR_RECIPE_SAVE_ERR ERR_NO_RECIPE_MANAGER_SET ERR_OK

Method ReadAndSaveRecipeAs

This method reads the current PLC values into the recipe. It subsequently saves the recipe to a specified recipe file. The contents of any existing file are overwritten.



Inputs (VAR_INPUT)

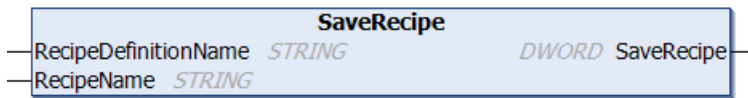
Name	Data type	Description
RecipeDefinitionName	STRING	Name of the recipe definition
RecipeName	STRING	Name of the recipe
FileName	STRING	Name of the file

Return value

Name	Data type	Description
ReadAndSaveRecipeAs	DWORD	Return value, possible values: ERR_RECIPE_DEFINITION_NOT_FOUND ERR_RECIPE_NOT_FOUND ERR_RECIPE_SAVE_ERR ERR_NO_RECIPE_MANAGER_SET ERR_OK

Method SaveRecipe

This method saves the recipe to a recipe file with a standard name. The standard name is <Recipe>.<Recipe definition>.<Recipe extension>. The contents of any existing file are overwritten.



Inputs (VAR_INPUT)

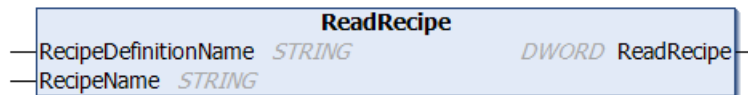
Name	Data type	Description
RecipeDefinitionName	STRING	Name of the recipe definition
RecipeName	STRING	Name of the recipe

Return value

Name	Data type	Description
SaveRecipe	DWORD	ERR_RECIPE_DEFINITION_NOT_FOUND ERR_RECIPE_NOT_FOUND ERR_RECIPE_SAVE_ERR ERR_NO_RECIPE_MANAGER_SET ERR_OK

Method ReadRecipe

This method reads the current PLC values into the recipe.



Inputs (VAR_INPUT)

Name	Data type	Description
RecipeDefinitionName	STRING	Name of the recipe definition
RecipeName	STRING	Name of the recipe

Return value

Name	Data type	Description
ReadRecipe	DWORD	ERR_RECIPE_DEFINITION_NOT_FOUND ERR_RECIPE_NOT_FOUND ERR_NO_RECIPE_MANAGER_SET ERR_OK Data server error from 16#2000 to 16#20FF Data source driver error from 16#2100 to 16#21FF

Method WriteRecipe

This method writes the recipe into the PLC variables.



Inputs (VAR_INPUT)

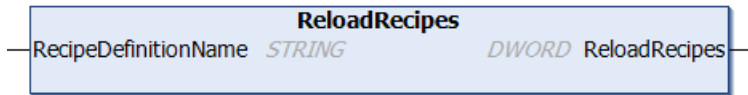
Name	Data type	Description
RecipeDefinitionName	STRING	Name of the recipe definition
RecipeName	STRING	Name of the recipe

Return value

Name	Data type	Description
WriteRecipe	DWORD	ERR_RECIPE_DEFINITION_NOT_FOUND ERR_RECIPE_NOT_FOUND ERR_NO_RECIPE_MANAGER_SET ERR_OK

Method ReloadRecipes

This method reads the list of recipes from a file system. Recipes with the standard name <Recipe>.<Recipe definition>.<Recipe extension> located in the path defined in the recipe manager are thereby considered.



Inputs (VAR_INPUT)

Name	Data type	Description
RecipeDefinitionName	STRING	Name of the recipe definition

Return value

Name	Data type	Description
ReloadRecipes	DWORD	ERR_RECIPE_DEFINITION_NOT_FOUND ERR_NO_RECIPE_MANAGER_SET ERR_OK

Method GetRecipeCount

This method returns the number of recipes in a recipe definition.



Inputs (VAR_INPUT)

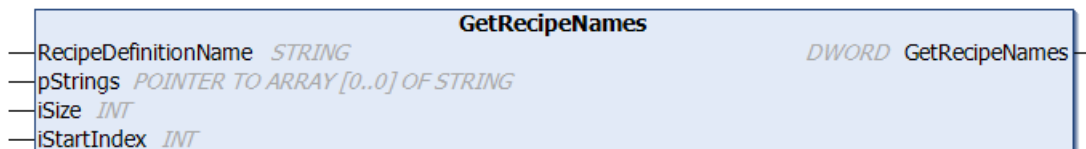
Name	Data type	Description
RecipeDefinitionName	STRING	Name of the recipe definition

Return value

Name	Data type	Description
GetRecipeCount	INT	ERR_RECIPE_DEFINITION_NOT_FOUND ERR_NO_RECIPE_MANAGER_SET ERR_OK

Method GetRecipeNames

This method returns the recipe names in a recipe definition.



Inputs (VAR_INPUT)

Name	Data type	Description
RecipeDefinitionName	STRING	Name of the recipe definition
pStrings	POINTER TO ARRAY [] OF STRING	Pointer to the array containing the recipe names.
iSize	INT	Number of elements in the STRING array
iStartIndex	INT	Start index Example: 1

Return value

Name	Data type	Description
GetRecipeNames	DWORD	Return value, possible values: ERR_RECIPE_DEFINITION_NOT_FOUND ERR_NO_RECIPE_MANAGER_SET ERR_OK

Example:

There are 50 recipes, for example. If you wish to generate a table that displays 10 recipe names simultaneously, you must define a STRING array:

```
strArr: ARRAY[0..9] OF STRING;
```

Corresponding to the iStartIndex you then obtain the recipe names for the specified range.

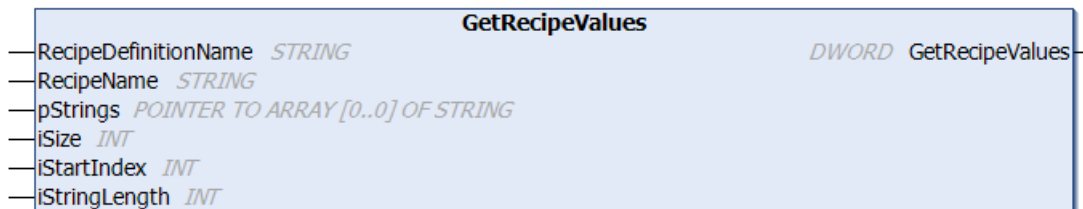
```
iStartIndex := 0; die Namen 0..9 werden zurückgegeben  
iStartIndex := 20; die Namen 20..29 werden zurückgegeben
```

The following applies in this example:

```
iSize := 10;
```

Method GetRecipeValues

This method returns the values of a recipe.



Inputs (VAR_INPUT)

Name	Data type	Description
RecipeDefinitionName	STRING	Name of the recipe definition
RecipeName		Name of the recipe
pStrings	POINTER TO ARRAY [] OF STRINGS	Pointer to a string in which the recipe values are stored.
iSize	INT	Size of the string array.
iStartIndex	INT	The start index.
iStringLength	INT	The length of the string in the array.

Return value

Name	Data type	Description
GetRecipeValues	DWORD	ERR_RECIPE_DEFINITION_NOT_FOUND ERR_RECIPE_NOT_FOUND ERR_NO_RECIPE_MANAGER_SET ERR_OK

Example:

There are 50 recipes, for example. If you wish to generate a table that displays 10 recipe values simultaneously, you must define a STRING array:

```
strArr: ARRAY[0..9] OF STRING;
```

Corresponding to the `iStartIndex` you then obtain the recipe values for a specified range.

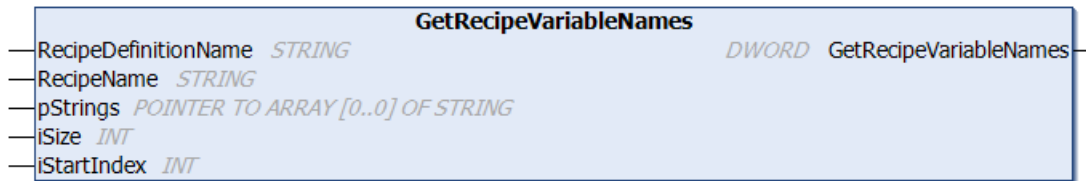
```
iStartIndex := 0; die Werte 0..9 werden zurückgegeben
iStartIndex := 20; die Namen 20..29 werden zurückgegeben
```

The following applies in this example:

```
iStringLength := 80;
iSize := 10;
```

Method GetRecipeVariableNames

This method returns the names of the recipe variables of a recipe.



Inputs (VAR_INPUT)

Name	Data type	Description
RecipeDefinitionName	STRING	Name of the recipe definition
RecipeName		Name of the recipe
pStrings	POINTER TO ARRAY [] OF STRINGS	Pointer to STRING data types in which the names of the recipe variables are stored.
iSize	INT	Size of the STRING array.
iStartIndex	INT	The start index.

Return value

Name	Data type	Description
GetRecipeVariableNames	DWORD	ERR_RECIPE_DEFINITION_NOT_FOUND ERR_RECIPE_NOT_FOUND ERR_NO_RECIPE_MANAGER_SET ERR_OK

Example:

There are 50 recipes, for example. If you wish to generate a table that displays 10 variable names of a recipe simultaneously, you must define a STRING array:

```
strArr: ARRAY[0..9] OF STRING;
```

Corresponding to the `iStartIndex` you then obtain the variable names of the recipe for the specified range.

```
iStartIndex := 0; die Namen 0..9 werden zurückgegeben
iStartIndex := 20; die Namen 20..29 werden zurückgegeben
```

The following applies in this example:

```
iSize := 10;
```

Method SetRecipeValues

This method sets the recipe values of a recipe.



Inputs (VAR_INPUT)

Name	Data type	Description
RecipeDefinitionName	STRING	Name of the recipe definition
RecipeName		Name of the recipe
pStrings	POINTER TO ARRAY [] OF STRINGS	Pointer to STRING data types in which the recipe values are stored.
iSize	INT	Size of the STRING array.
iStartIndex	INT	The start index.

Return value

Name	Data type	Description
SetRecipeValues	DWORD	ERR_RECIPE_DEFINITION_NOT_FOUND ERR_RECIPE_NOT_FOUND ERR_NO_RECIPE_MANAGER_SET ERR_OK

Example:

There are 50 recipes, for example. If you wish to generate a table that sets 10 recipe values simultaneously, you must define a STRING array:

```
strArr: ARRAY[0..9] OF STRING;
```

Corresponding to the iStartIndex you can then set the recipe values for a specified range.

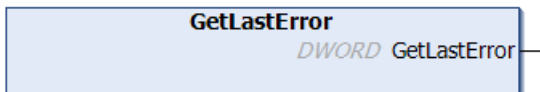
```
iStartIndex := 0; Die Werte 0..9 werden gesetzt  
iStartIndex := 20; Die Werte 20..29 werden gesetzt
```

The following applies in this example:

```
iStringLength := 80;  
iSize := 10;
```

Method GetLastError

This method returns the last error of the preceding operation.

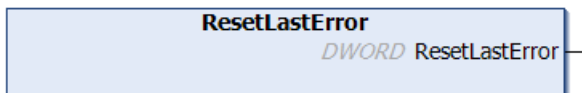


Return value

Name	Data type	Description
GetLastError	DWORD	Return value, possible values: ERR_NO_RECIPE_MANAGER_SET ERR_OK

Method ResetLastError

This method resets the last error.

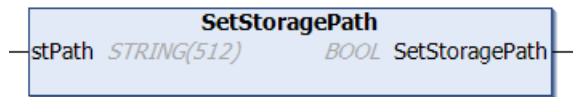


Return value

Name	Data type	Description
ResetLastError	DWORD	ERR_NO_RECIPE_MANAGER_SET ERR_OK

Method SetStoragePath

This method is for setting the storage path for the recipe file. It overwrites the path specification in the dialog Storage of the recipe manager.



Inputs (VAR_INPUT)

Name	Data type	Description
stPath	STRING	File path, example: <ul style="list-style-type: none"> D:/recipefiles/

Return value

Name	Data type	Description
stPath	BOOL	TRUE (path was set) FALSE Possible errors: ERR_OK ERR_NO_RECIPE_MANAGER_SET

Return values

The return values of the functions described above are contained in the GVL ReturnValues.

These are InOut constants of the data type UDINT.

Name	Initialization value	Comment
ERR_OK	16#0	The operation was carried out successfully.
ERR_FAILED	16#1	The operation failed.
ERR_PARAMETER	16#2	Incorrect parameter
ERR_NOTINITIALIZED	16#3	The data server object is not initialized. The data server is required if the recipe management is used in combination with TwinCAT HMI.
ERR_NOTIMPLEMENTED	16#C	The data server does not implement the interface IDataServer4 that is required if the recipe management is used in combination with TwinCAT HMI.
ERR_NO_OBJECT	16#10	Not all variables of a recipe definition can be written via the data server. Only the valid variables are written.
ERR_NOMEMORY	16#11	The data server does not have sufficient memory.
ERR_RECIPE_FILE_NOT_FOUND	16#4000	The recipe file was not found.
ERR_RECIPE_MISMATCH	16#4001	The contents of the recipe file do not match the current recipe. This error is only output if the storage type is textual (see editor Recipe Manager , tab Storage , Storage Type) and if a variable name in the file does not correspond to the variable name in the recipe definition. The recipe file is not loaded if this error occurs. Possible causes: A variable was removed from the recipe definition in the project.
ERR_RECIPE_SAVE_ERR	16#4002	The storage procedure failed. Possible causes <ul style="list-style-type: none"> • The file could not be created or opened because the hard disk is full. • The configured file path does not exist (see editor Recipe Manager, tab Storage, File Path). • The configured file extension is not permitted by the runtime system (see editor Recipe Manager, tab Storage, File Extension).
ERR_RECIPE_NOT_FOUND	16#4003	The recipe does not exist.
ERR_RECIPE_DEFINITION_NOT_FOUND	16#4004	The recipe definition does not exist.
ERR_RECIPE_ALREADY_EXIST	16#4005	The recipe already exists in the recipe definition. Create a new recipe under a different name.
ERR_NO_RECIPE_MANAGER_SET	16#4006	The global recipe manager has not been created. Possible cause: <ul style="list-style-type: none"> • The option Recipe management in the PLC is not set in the editor of the recipe manager, tab General of the current PLC project.
ERR_RECIPE_NOT_ALL_VARIABLES_WERE_LOADED	16#4007	The recipe definition contains more variables than the recipe file. In this case the variable values of the recipe file will be written in any case. This is only for information, not actually an error.
ERR_RECIPE_NOMEMORY	16#4008	The recipe definition has no free memory to create a new recipe. Possible causes <ul style="list-style-type: none"> • The option Save recipe changes to recipe files automatically is not set in the editor of the recipe manager, tab General of the current PLC project. • In this case only 50 recipes are possible per recipe definition. The error cannot occur if the option Save recipe changes to recipe files automatically is set. If the hard disk is full the error ERR_RECIPE_SAVE_ERR will be output.
ERR_RECIPE_MANAGER_LOCKED_DURING_ONLINE_CHANGE	16#4009	The recipe manager was blocked during the online change. Possible causes: Some of the RecipeMan commands that should have been executed during an online change were not executed.
ERR_SOURCE_EXHAUSTED	16#40A0	Used for UTF8 Helper
ERR_TARGET_EXHAUSTED	16#40A1	Used for UTF8 Helper
ERR_SOURCE_ILLEGAL	16#40A2	Used for UTF8 Helper

10.3 Error analysis with core dump

A core dump is a dump of the PLC project data.



Available from TC3.1 Build 4024.11



Core dump can only be used with the associated compile info file

If you archive or save a core dump file, please note that the associated project and associated compile info file (*.compileinfo file, which is stored, for example when creating the project, in the "_CompileInfo" folder) must be present in order to load a core dump. If this is not the case, TwinCAT cannot use the dump later.

Please also note here the setting options on the [Settings tab](#) [▶ 926]. With the help of the **Core Dump** setting, you can configure whether the core dump file, which may be located in the project directory, is to be saved together with the available compile info files in a TwinCAT file archive.

Generating a Core Dump

1) Automatic generation

In case of an exception error, runtime systems automatically store a core dump on the target system if the associated PLC project is not currently logged in to a development environment. By default, this core dump is stored as a *.core file in the boot folder of the target system. Further information on the (adjustable) storage path can be found below. The automatic generation of a core dump also occurs if the exception error occurs within the code of an FB_init/FB_reinit/FB_Exit method (from TC3.1 Build 4024.25).



Warning message when core dump is available on target system

If you login a PLC project and there is a core dump on the selected target system, a warning is displayed in the message window indicating that a core dump is available on the device.

If the warning is no longer to be displayed when logging in the project (e.g. because you have completed the analysis of this core dump file or because you have archived the file somewhere else), you can delete the respective core dump file from the target system.

2) Manual generation

In online mode, you can also explicitly generate a core dump if the PLC project is currently at a breakpoint or an exception has occurred. In this case TwinCAT stores the core dump file only in the project directory and not on the target system. The core dump file generated is saved directly in the PLC project directory (<PLC_project_name>.<PLC_project_GUID>.core).

Loading a core dump

You can load the core dump from the target system into the project directory in online mode (using the [Command Generate core dump](#) [▶ 951]). In addition, you can load a core dump from the project directory into the project in offline mode (using the [Command Load core dump](#) [▶ 952]). You will then receive an online view of the PLC project with the data and values at the time of the exception or at the time of generating the core dump file.



In the online view that TwinCAT generates when loading the core dump into the project, menu commands appear as available that are not effective in this state. If you select such a command, you will receive a corresponding message.

In addition, you can only close the core dump view using the [command Close core dump](#) [▶ 953]. The Logout command is not effective in this view.

Archiving

If the core dump file located in the project directory is to be contained in the project archive, activate the "Core Dump" option during the configuration of the file archive content (see documentation Reference User Interface > Project > PLC project settings > [Settings tab \[► 926\]](#)).

Storage path for automatic creation



Available from TwinCAT 3.1 Build 4026

Default:

By default, automatically generated core dumps are stored as *.core files in the boot folder of the target system. By default, this is located at:

- < TC3.1.4026.0: C:\TwinCAT\3.1\Boot\Plc
- >=TC3.1.4026.0: C:\ProgramData\Beckhoff\TwinCAT\3.1\Boot\Plc

Individual adjustment:

If desired, the storage path of the automatic core dump creation can be adjusted. To do this, the desired storage path must be entered in the Windows registry of the target system.

1. Open the following registry path on the target system:
 - 32-bit system: "HKEY_LOCAL_MACHINE\SOFTWARE\Beckhoff\TwinCAT3\Plc"
 - 64-bit system: "HKEY_LOCAL_MACHINE\SOFTWARE\WOW6432Node\Beckhoff\TwinCAT3\Plc"
2. If the "Plc" folder does not yet exist, add it.
3. Add the key "**CoreDumpDir**" as a **string** and enter the desired file path for the automatic core dump creation as the value.
4. Set TwinCAT in Config.
 - ⇒ The path is activated the next time the TwinCAT state changes to Run.

Two exemplary workflows are described below.

Load core dump from target system to project for analysis

Requirement:

- ✓ With the development environment, you have opened the project that produced an exception error on the target system. The PLC project is in online mode in the development environment.
1. Use the [Command Generate core dump \[► 951\]](#) to load the core dump from the target system.
 - ⇒ TwinCAT copies the core dump file with the name <PLC project name>.<PLC project GUID>.core into the local PLC project directory.
 2. Log the PLC project out.
 3. Use the [Command Load core dump \[► 952\]](#) to load the desired core dump into the project.
 - ⇒ TwinCAT displays an online view of the PLC project. You can see the variable values and call stack at the time of the error.
 4. After completing the core dump analysis, select the [Command Close core dump \[► 953\]](#).
 - ⇒ TwinCAT closes the core dump view of the PLC project. The development environment reappears with the views of normal offline operation.

Manual generation of a core dump of the logged-in PLC project

- ✓ A PLC project is in online mode in the development environment. The PLC project is currently at a breakpoint or an exception error has occurred.
1. Select the [Command Generate core dump \[► 951\]](#).
 - ⇒ TwinCAT generates a new core dump and creates the file with the name <PLC_project_name>.<PLC_project_GUID>.core in the local PLC project directory.

See also:

- Documentation TC3 User Interface: Reference User Interface > PLC > Core Dump >
 - [Command Generate core dump \[▶ 951\]](#)
 - [Command Load core dump \[▶ 952\]](#)
 - [Command Close core dump \[▶ 953\]](#)

10.4 PLC operation control



Available from TC3.1 Build 4026



It is your responsibility to ensure that runtime system services are activated in safe application states and deactivated only in critical ones.

A plant or project can enter a sensitive state at runtime, where disruptive actions can endanger the entire machine or plant. However, in this state you can suppress certain commands and prevent dangerous actions. The global data type `PlcAppSystemInfo` is available for this purpose.

Examples of TwinCAT commands whose execution can be suppressed:

- Write values, force values
- Set a breakpoint
- Download, Online Change

If a runtime system service is requested at runtime of the project, but it is currently deactivated, you will receive a message about this in TwinCAT.

Using `PlcAppSystemInfo` for operation control

You can activate or deactivate operations using the `_AppInfo.Flags` variable of `DWORD` type. `_AppInfo` is an instance of the `PlcAppSystemInfo` type.

Operation	Bit of <code>_AppInfo.Flags</code>	Sample access
Deactivate Write values	Bit 0	<code>_AppInfo.Flags.0 := TRUE;</code>
Deactivate Force values	Bit 1	<code>_AppInfo.Flags.1 := TRUE;</code>
Deactivate Set a breakpoint	Bit 2	<code>_AppInfo.Flags.2 := TRUE;</code>
Deactivate Download	Bit 3	<code>_AppInfo.Flags.3 := TRUE;</code>
Deactivate Online Change	Bit 4	<code>_AppInfo.Flags.4 := TRUE;</code>

See also:

- [Bit Access to Variables \[▶ 752\]](#)

Also see about this

- 📖 [Bit Access to Variables \[▶ 752\]](#)

11 Updating the PLC project on the PLC

Situation: You are loading a PLC project onto the controller that differs from the project already located there. In this case, a message window appears in which you can select how you want to transfer the modified PLC project to the controller: download or online change.

- A download leads to a new compilation of the PLC program. In addition to a syntax check, program code is generated and loaded onto the controller. This will stop the running program. A download is the recommended type of data transfer, since it always leads to a defined output state as a result of the program stop and the reinitialization.
- During an online change, only the modified parts are reloaded into the controller. A running program is not stopped. The online change option should only be used after minor modifications of the PLC project. After more extensive modifications, the program behavior cannot be predicted reliably.

Revision overview

Possibility 1:

The message window described above also contains a **Details** button. Use this button to open the **Application information** window, which allows you to check the differences between the current PLC project and the PLC project on the controller. This involves comparing the number of function blocks, the data and the storage locations.

The Application information window contains a brief description of the differences, for example:

- Declaration of MAIN changed
- Variable fbMyNewInstance inserted in MAIN
- Number of methods/actions of FB_Sample changed

If the setting **Download application info** (PLC project properties, [category Compile \[► 910\]](#)) is enabled, an option for an extended check of the differences between the current PLC project and the PLC project on the controller is available. The difference in the extended check option is that the **Application information** window contains an additional **Online** comparison tab, which shows a tree comparison view. This will tell you which POU's have been changed, deleted or added. The additional tab appears when you execute the blue underlined command in the lower area of the application information window ("Application not current. Generate code now to display the online comparison?").

This tree comparison view provides only a rudimentary overview. For a detailed comparison of the current PLC project and the PLC project on the controller, please see option 2.

Possibility 2:

Requirement: The source code files of the PLC project were also transferred to the target system (can be configured via the [Settings tab \[► 926\]](#) in the PLC project settings).

The TwinCAT Project Compare Tool enables a detailed comparison of the current TC3 project and the TC3 project on the controller ([Command Compare <TwinCAT project name> with the target system... \[► 1065\]](#)). This opens a comparison tree view, which allows the desired PLC editors to be opened in a comparison window. Differences are highlighted in color. It is also possible to merge the source code of the target system project (or parts of the source code).

See also:

- Reference user interface documentation: [Command Download \[► 955\]](#)
- Reference user interface documentation: [Command Login \[► 957\]](#)
- Reference user interface documentation: [Command Online Change \[► 955\]](#)

11.1 Performing an Online Change

TwinCAT automatically offers you an online change if you log in with a PLC project that already exists on the controller but has been modified since the last download in the programming system. During this process, only the modified parts are reloaded into the controller. A program that is running on the controller is not stopped during the online change.

⚠ WARNING

Damage to property and persons due to unexpected behavior of the machine or system

An online change modifies the running application program and does not cause a restart. Depending on the machine being controlled, the machine or workpieces may be damaged or the health and life of people may be endangered.

- Make sure that the new program code results in the desired behavior of the controlled system.

● Project-specific initializations

i When an online change is performed, the project-specific initializations (homing etc.) are not executed because the machine retains its status. For this reason, the new program code may not have the desired effect.

● Major changes in the download code

i If the online change causes significant changes in the download code (e.g. shifting of variables is required), a dialog provides information about the effects and allows you to cancel the online change.

● Fast online change

i For small changes (e.g. in the implementation section, with no shifting of variables required), a "fast online change" is performed. In this case, only the modified function block is compiled and reloaded. In particular, no initialization code is generated in this case. This also means that no code for initializing variables with the attribute 'init_on_onlchange' is generated. Usually this will not have any effect, since the attribute tends to be used to initialize variables with addresses, but a variable cannot change its address during a fast online change.

To ensure the init_on_onlchange attribute is applied to the entire application code, deactivate fast online change for the PLC project using the no_fast_online_change compiler definition. To this end, insert the definition in the Compile category [▶ 910] of the PLC project properties.

● The attribute 'init_on_onlchange' has no effect for individual FB variables

i The Attribute 'init_on_onlchange' [▶ 807] only applies to global variables, program variables and local static variables of function blocks.

To reinitialize a function block during an online change, the function block instance must be declared with the attribute. The attribute is not evaluated for a single variable in a function block.

Pointer variables

Pointer variables retain their value from the last cycle. If a pointer points to a variable that has been resized by the online change, the value is no longer delivered correctly. Make sure that pointer variables are reassigned in each cycle.

Monitoring functions

After removing implicit monitoring functions such as CheckBound, no online change is possible, only a download. A corresponding message is issued.

See also:

- TC3 User Interface documentation: Command Online Change [▶ 955]

Online change on login

- ✓ The connection settings of the controller are set correctly. The application programs in the project and on the controller are identical. The program is running on the controller. The PLC project is logged out.

1. Change your PLC project.
2. Select the command **Login** in the **PLC** menu or in the **TwinCAT PLC Toolbar Options**.
 - ⇒ A dialog appears indicating that the application was changed since last download.
3. Select the option **Login with online change** and click **OK**.
 - ⇒ The change is loaded onto the controller. The program running on the controller is not stopped. The PLC project is logged in.

See also:

- TC3 User Interface documentation: [Command Login \[► 957\]](#)

Online change in logged-in state (online mode)

- ✓ The connection settings of the controller are set correctly. The application programs in the project and on the controller are identical. The program is running on the controller. The PLC project is logged in.
1. Select an object in the PLC project tree. Ideally a POU or GVL in this case.
 2. In the context menu select the command **Edit Object (Offline)**.
 - ⇒ The object opens in the editor.
 3. Change the object. for example by declaring a new variable or changing a value assignment.
 4. Select the command **Online Change** in the **PLC** menu.
 - ⇒ You will be asked whether you really want to carry out the online change.
 5. Confirm the dialog with **Yes**.
 - ⇒ The change is loaded onto the controller.

See also:

- TC3 User Interface documentation: [Command Edit object \(offline\) \[► 886\]](#)
- TC3 User Interface documentation: [Command Online Change \[► 955\]](#)

What prevents an online change?

After certain TwinCAT actions an online change on a controller is no longer possible. After that, a [download \[► 253\]](#) of the project is always required. A typical case are the actions **Clean** and **Clean all**, which delete the compilation information stored during the last download. But there are also "normal" editing actions that result in an online change not being possible the next time you log in. The following actions can prevent an online change:

Check functions	Activating or removing a check function [► 163] (CheckBounds, CheckRange, CheckDiv etc.). Changing the interface of a check function (including inserting or deleting local variables).
Task configuration	Changing the configuration settings.
Project settings	Category Compile [► 910]: Changing the compiler defines in the Settings section (replace constants) Category Common [► 906]: Minimize ID changes in TwinCAT files.
Function block properties	Change of option External implementation
Function block	Changing the basic block of a function block (EXTENDS [► 191] FB_Base), or inserting or deleting such a basic block. Changing the interface list (IMPLEMENTS [► 198] I_Sample). Exception: Adding a new interface at the end of the list.
Data type	Changing the data type of a variable from one user-defined data type to another user-defined data type (e.g. from TON to TOF). Changing the data type from a user-defined data type to a basic data type (e.g. from TON to TIME). Note: As a workaround, always change the name of the variable at the same time as the data type. As a result, the variable is initialized as a new variable, and the old variable is removed. An online change is then possible.

11.2 Execution of a Download

Downloading the PLC project causes the active project to be compiled. In addition to a syntax check, program code is generated and loaded onto the controller. A program that is running on the controller is stopped during the download.



All variables except persistent variables are reinitialized during the download.

See also:

- TC3 User Interface documentation: [Command Download](#) [► 955]

Download at login

- ✓ The connection settings of the controller are set correctly. The applications in the project and on the controller are identical. The program is running on the controller. The PLC project is logged out.
1. Change your PLC project.
 2. Select the command **Login** in the **PLC** menu or in the **TwinCAT PLC Toolbar Options**.
⇒ A dialog appears indicating that the application was changed since last download.
 3. Select the option **Login with download** and click **OK**.
⇒ The program that is running on the controller is paused while the change is loaded onto the controller. The PLC project is logged in.

See also:

- TC3 User Interface documentation: [Command Login](#) [► 957]

Download in logged-in state (online mode)

- ✓ The connection settings of the controller are set correctly. The applications in the project and on the controller are identical. The program is running on the controller. The PLC project is logged in.
1. Select an object in the PLC project tree. Ideally a POU or GVL in this case.

2. In the context menu select the command **Edit Object (Offline)**.
 - ⇒ The object opens in the editor.
3. Change the object. for example by declaring a new variable or changing a value assignment.
4. Select the command **Download** in the **PLC** menu.
 - ⇒ You will be asked whether you really want to carry out the **Download** operation.
5. Confirm the dialog with **Yes**.
 - ⇒ The program that is running on the controller is paused while the change is loaded onto the controller.

See also:

- TC3 User Interface documentation: [Command Edit object \(offline\) \[► 886\]](#)
- TC3 User Interface documentation: [Command Download \[► 955\]](#)

12 Using a stand-alone PLC project

A stand-alone PLC project manages a conventional PLC project (PLC project embedded in the TwinCAT project) in a separate project type. Programming is the same as for an embedded PLC project. In contrast to this, the PLC instance is not directly available in the project, but can only be configured within the TwinCAT project after integration. The TwinCAT 3 type system is also available in the stand-alone PLC project for managing project-based data types or configuring the TwinCAT 3 event logger.

Stand-alone PLC projects thus make it possible to separate system, motion and I/O configuration from PLC development at project level. This results in two separate projects: the stand-alone PLC project and the TwinCAT project.

The two projects can be managed in a common solution or in separate solutions. In order to link the PLC module instance in the TwinCAT project with the associated components at system, motion and I/O level, the TMC file compiled in the stand-alone PLC project is integrated.

This procedure enables, for example, the strict separation of system configuration and PLC programming, the integration of a PLC instance into different system configurations or the download of the PLC project runtime data under certain conditions from the PLC project without exception.

Requirement

Stand-alone PLC projects can be used from TwinCAT version 3.1.4022.0.

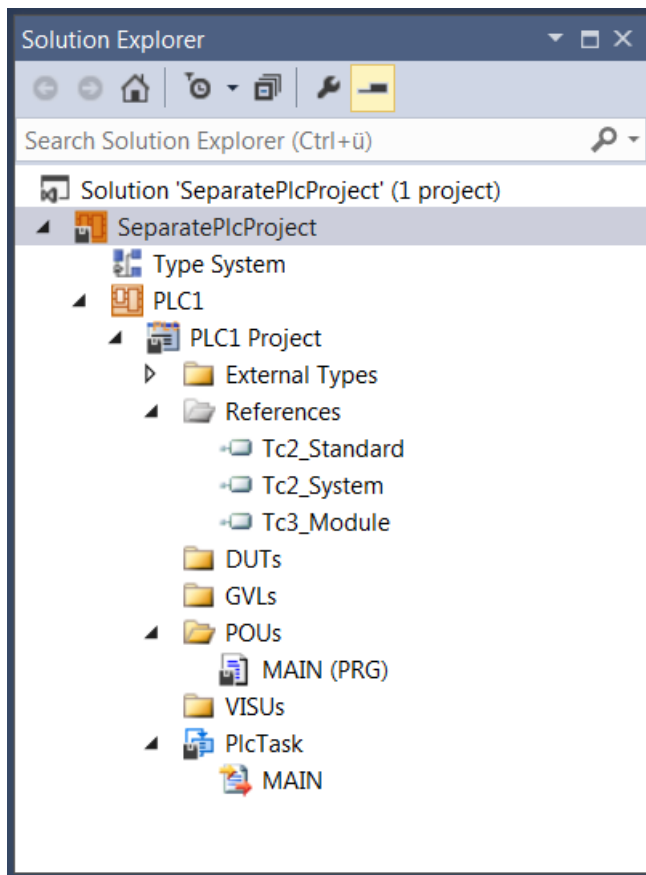
See also:

- [Creating a standard project](#) [► 54]
- [Type system](#)

12.1 Creating a stand-alone PLC Project

1. Select the command **New > Project** in the menu **File**.
 - ⇒ The dialog **New Project** opens.
2. Select the TwinCAT PLC template **TwinCAT PLC project**.
3. Give the PLC project a name (e.g. "SeparatePlcProject") and select a storage location and a solution.
4. Confirm the dialog with **OK**.
 - ⇒ A new project folder opens in the **Solution Explorer**. The project name "SeparatePlcProject" appears in the title bar of the main window.
5. Mark the PLC object in the project tree and select the command **Add New Item...** in the menu **Project** or in the context menu.
6. Select the **Standard PLC Project** in the **Plc Templates** category and enter a name (e.g. "PLC1").
7. Quit the dialog with **Add**.

⇒ The following structure is created in the view **Solution Explorer**:



⇒ Below the PLC project object (PLC1), the base objects of a standard PLC project and the type system appear automatically. In contrast to the standard PLC project, which is embedded in a TwinCAT project, no PLC instance is created.

See also:

- [Creating a standard project](#) [► 54]

12.2 Integrating the stand-alone PLC project into a TwinCAT project

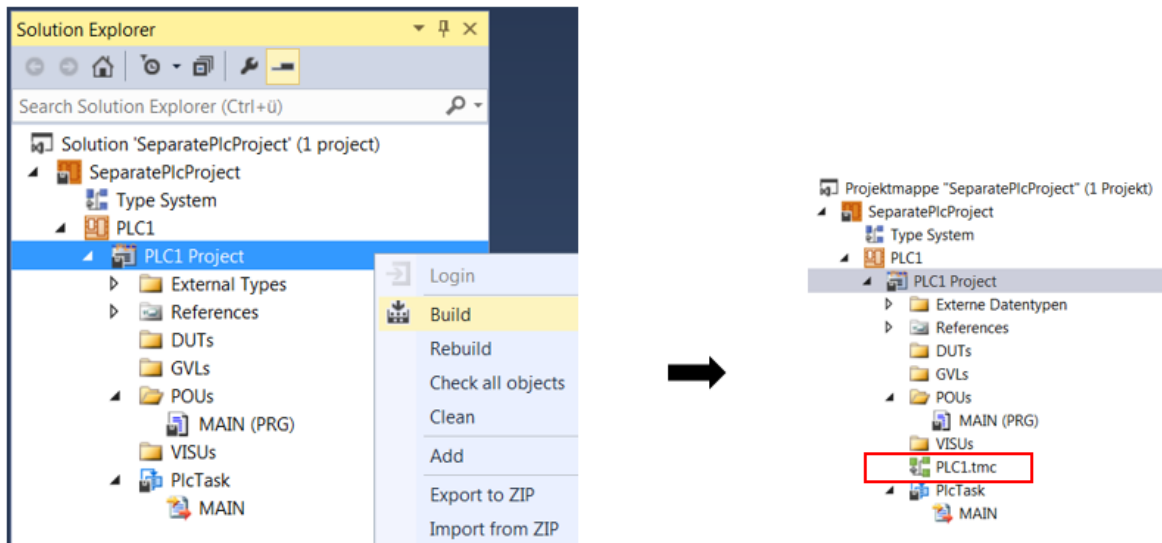
A stand-alone PLC project is integrated into a TwinCAT project via a TMC file. This is generated when the stand-alone PLC project is created and added to the TwinCAT project.

12.2.1 Creating a TMC file and adding it to the TwinCAT project

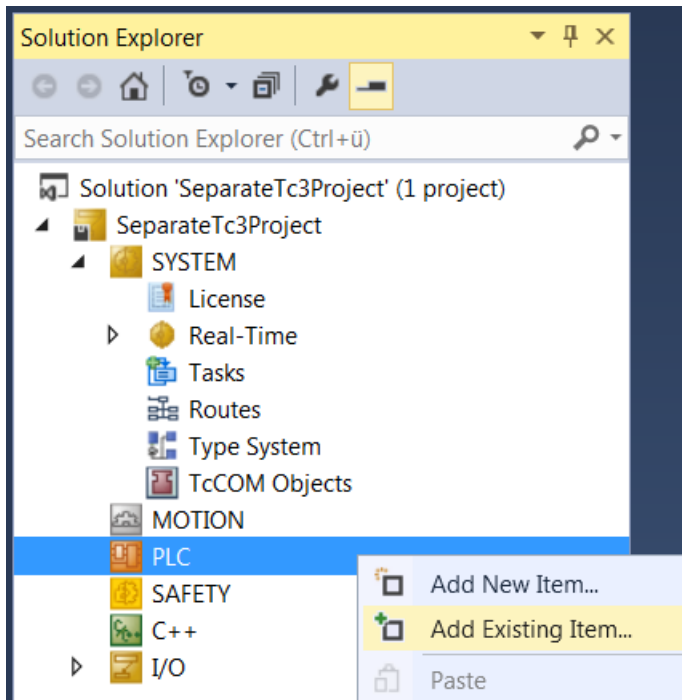
The following section describes how to create a TMC file and add it to the TwinCAT project. Multiple addition of the TMC file in the same TwinCAT project is not permitted.

- ✓ A stand-alone PLC project has been created (e.g. "SeparatePlcProject"), which contains a PLC project (e.g. "PLC1").
 - ✓ A TwinCAT project has been created into which the stand-alone PLC project is to be integrated (e.g. "SeparateTc3Project").
1. Open the stand-alone PLC project and select the PLC project object ("PLC1 Project") in the PLC project tree.
 2. Select the command **Build (Build <PLC Project Name>)** from the context menu or the **Build** menu.

⇒ The PLC project is compiled and checked for errors. If the compilation is successful, the TMC file is created.

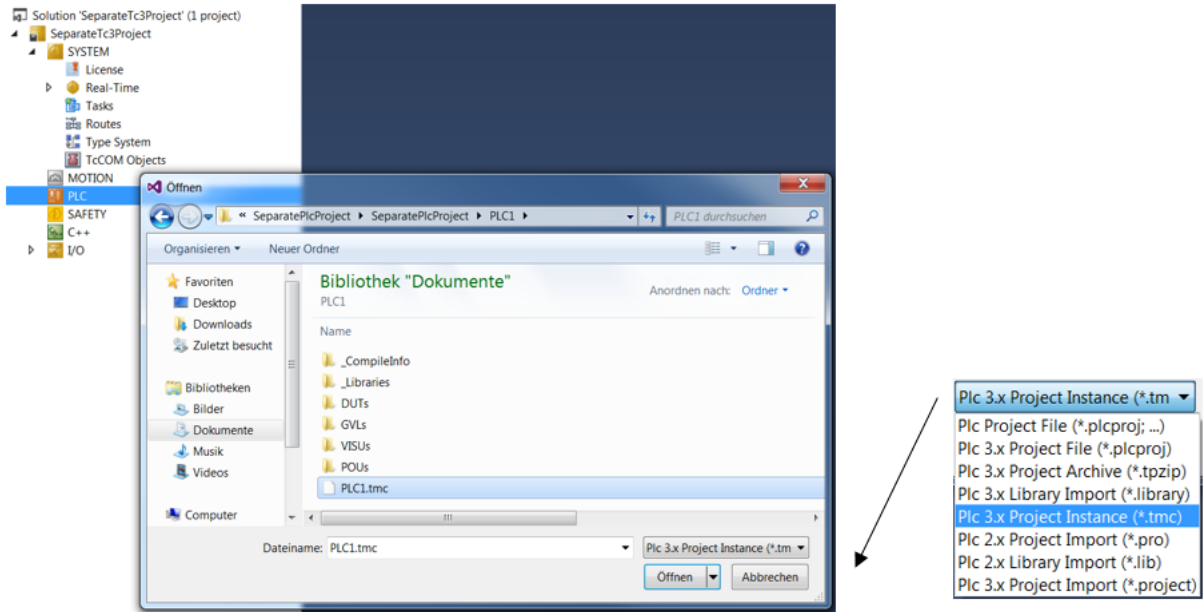


3. Open the TwinCAT project and select the PLC object in the TwinCAT project tree.
4. Select **Add Existing Item** from the context menu or the **Project** menu.

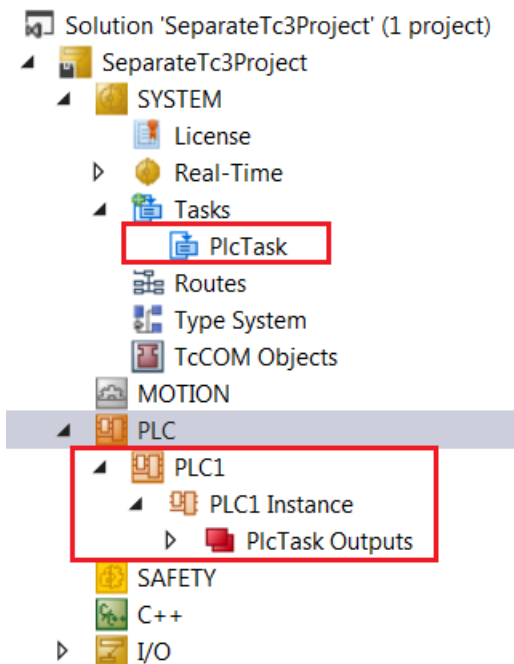


⇒ The standard dialog for opening a file appears.

5. Select the TMC file and click **Open** to add it to the project.

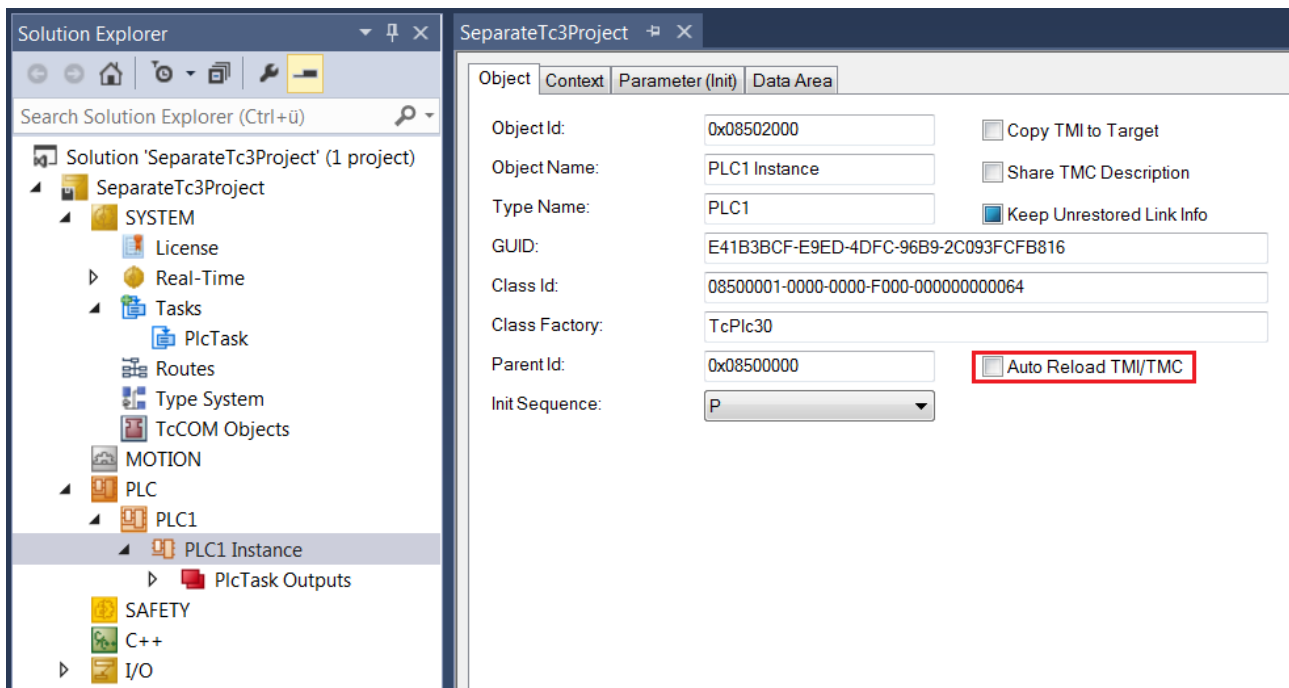


⇒ The PLC instance of the stand-alone PLC project and the referenced tasks are added to the TwinCAT project. If a system task with the name of the referenced task of the PLC project already exists in the TwinCAT project, no new task is added. Instead, the existing system task is linked to the added PLC instance.



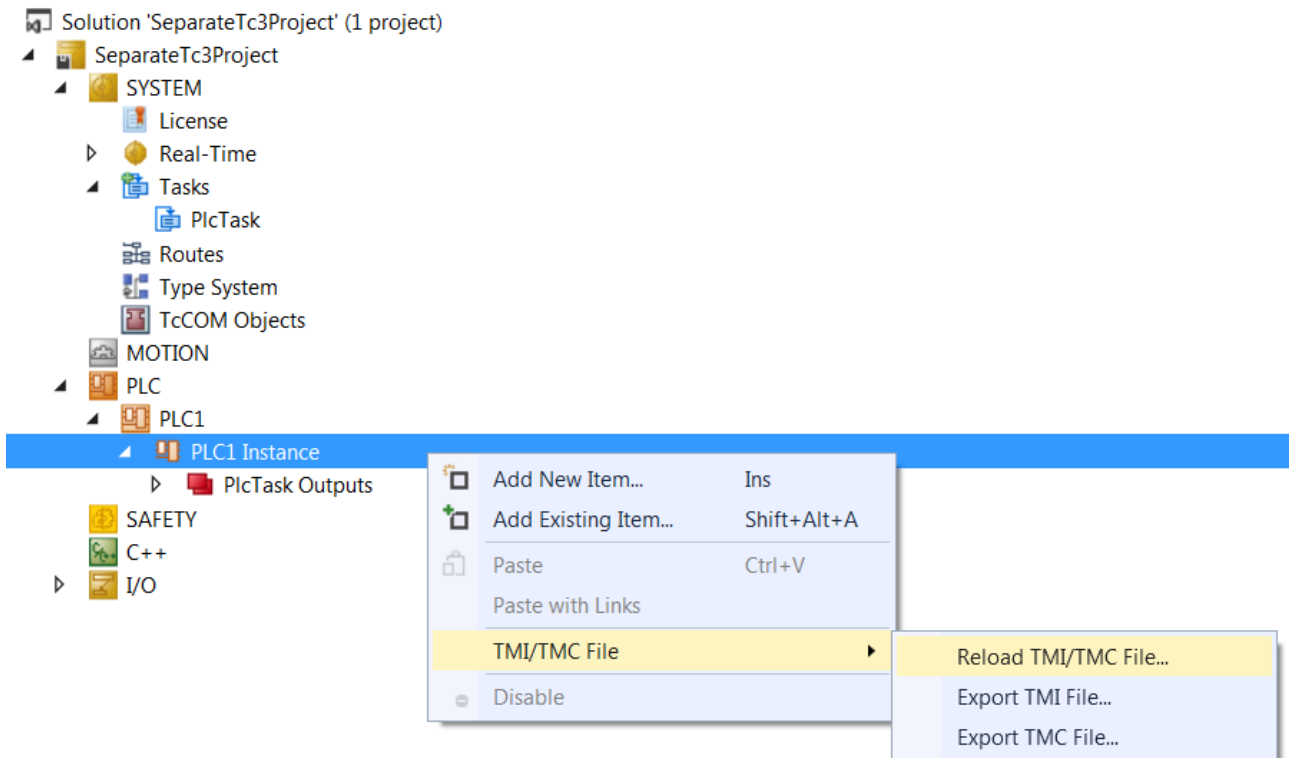
Automatic reloading of the TMC file

In the settings of the PLC instance, you can specify that the TMC file should automatically be reloaded when the stand-alone PLC project is changed. Double-click on the PLC instance of the stand-alone PLC project in the project tree to open the PLC instance settings in the editor. On the **Object** tab, tick the **Auto Reload TMI/TMC** check box.



Manual reloading of the TMC file

To manually reload the TMC file in the TwinCAT project when the stand-alone PLC project changes, select the PLC instance of the stand-alone PLC project in the project tree and select **TMI/TMC File > Reload TMI/TMC File** from the context menu.

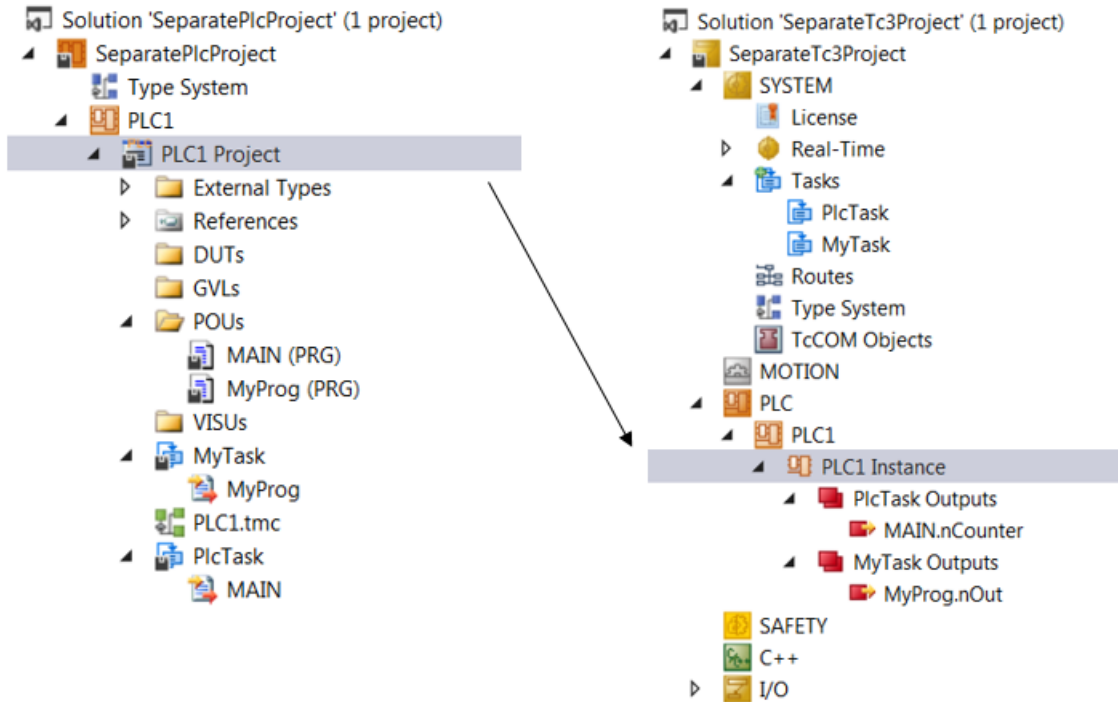


12.2.2 Assigning a task

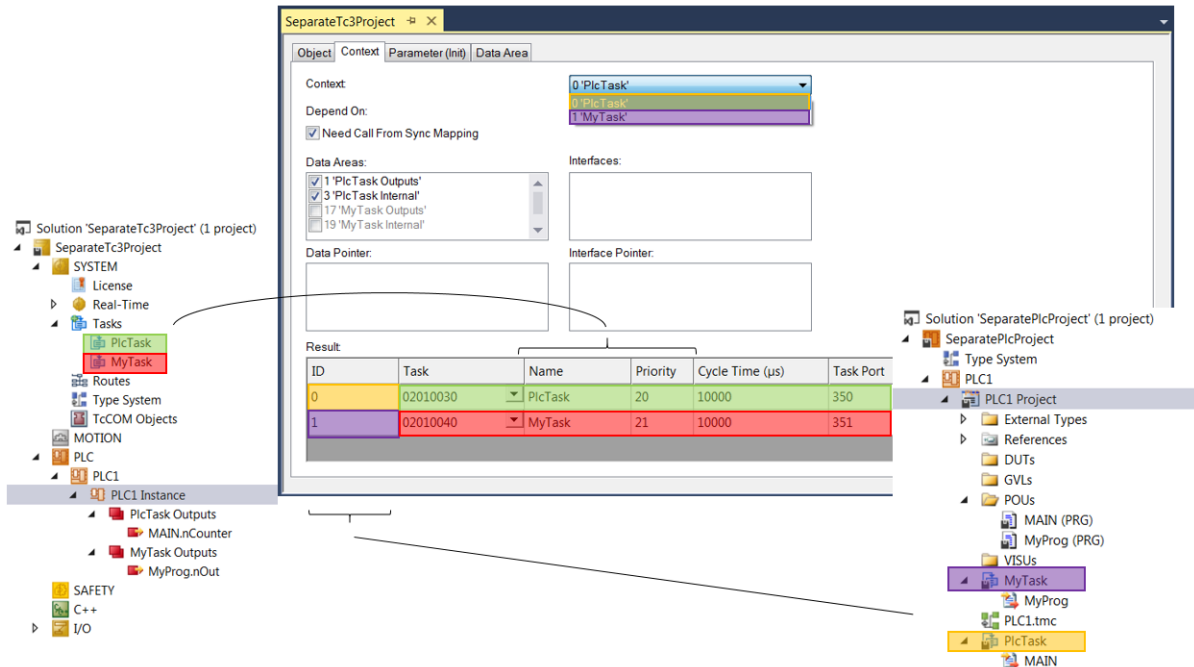
The following section describes how to assign the task references created for the stand-alone PLC project to the system tasks.

- ✓ Various task references have been generated for the stand-alone PLC project (e.g. "PlcTask" and "MyTask"), which define the processing of the MAIN and MyProg program blocks.

- ✓ The PLC instance of the stand-alone PLC project is integrated into the TwinCAT project.



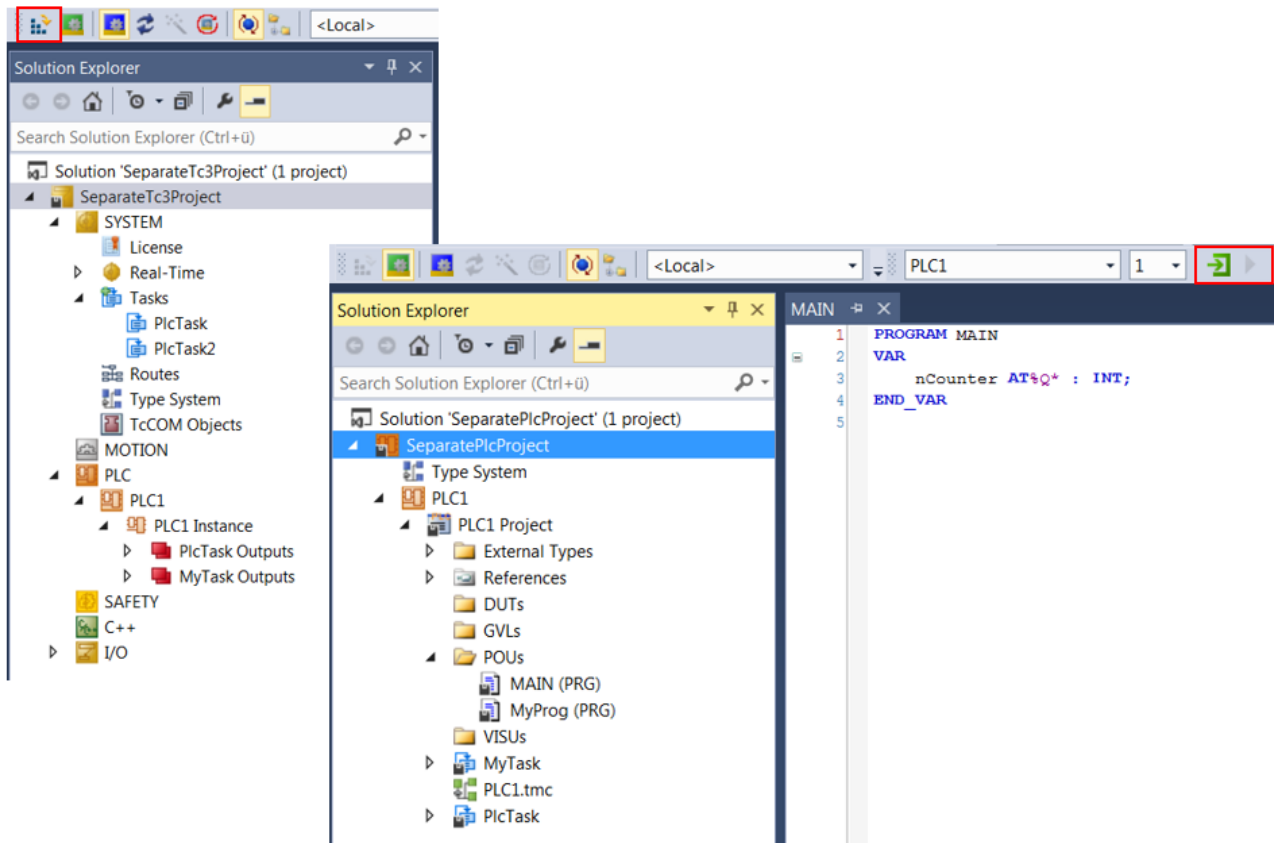
1. Double-click on the integrated PLC instance in the TwinCAT project tree to open the PLC instance settings in the editor.
2. Select the **Context** tab.
3. Assign the task references to the existing system tasks. (In this example, the task references are assigned to the system tasks of the same name.)



12.2.3 Loading program code, logging in and starting the PLC

- ✓ A stand-alone PLC project is integrated into a TwinCAT project.
1. Activate the TwinCAT project configuration. To do this, select the command **Activate Configuration** from the toolbar.

2. Log into the PLC in the stand-alone PLC project and start the PLC.



12.3 Best practice

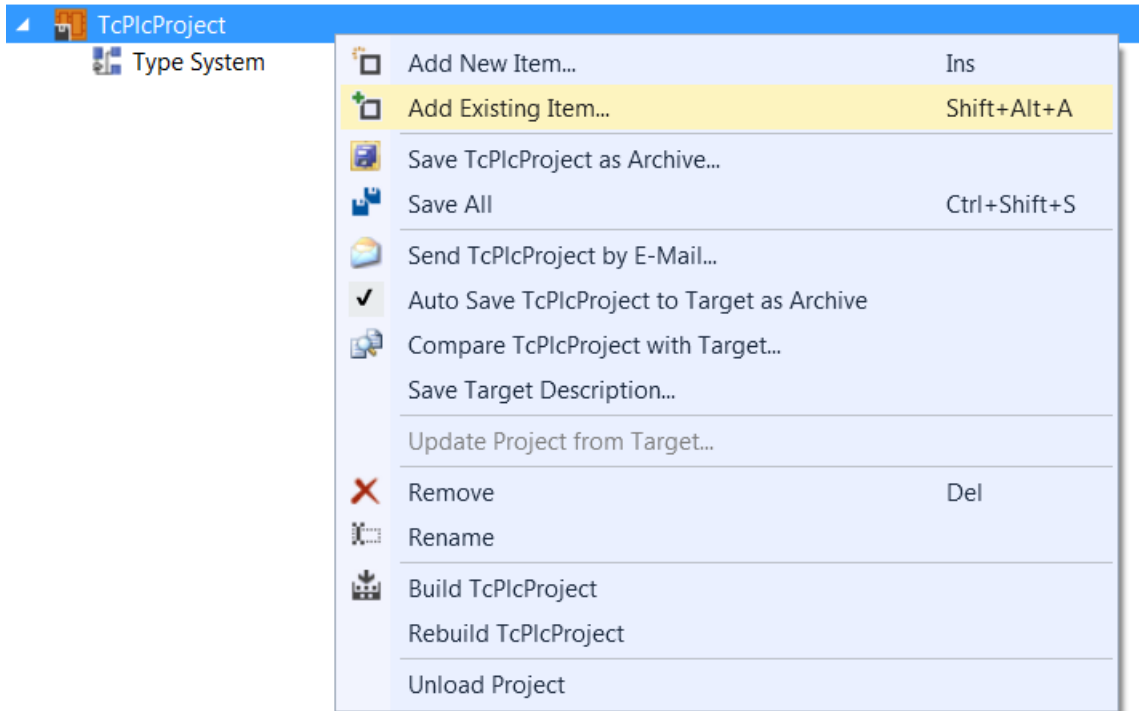
Overview

- [Converting the embedded PLC project of an existing TwinCAT project into a stand-alone PLC project \[► 261\]](#)
- [Converting a stand-alone PLC project into an embedded PLC project \[► 264\]](#)

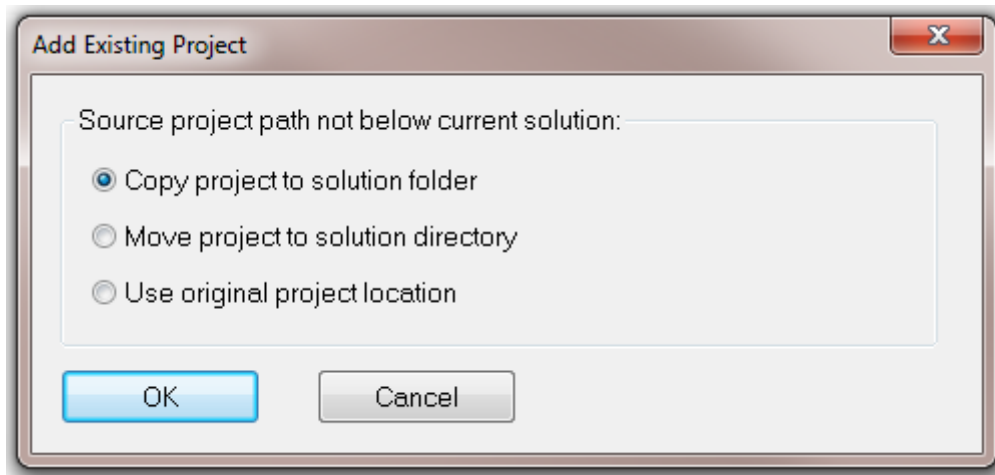
Converting the embedded PLC project of an existing TwinCAT project into a stand-alone PLC project

1. Create a project of type "stand-alone PLC project". Do not create a PLC project.

2. Add the embedded PLC project of the existing TwinCAT project to the stand-alone PLC project. To do this, click on **Add Existing Item...** in the context menu of the PLC project and select the .plcproj file of the embedded PLC project in the standard dialog that opens. Confirm the dialog.

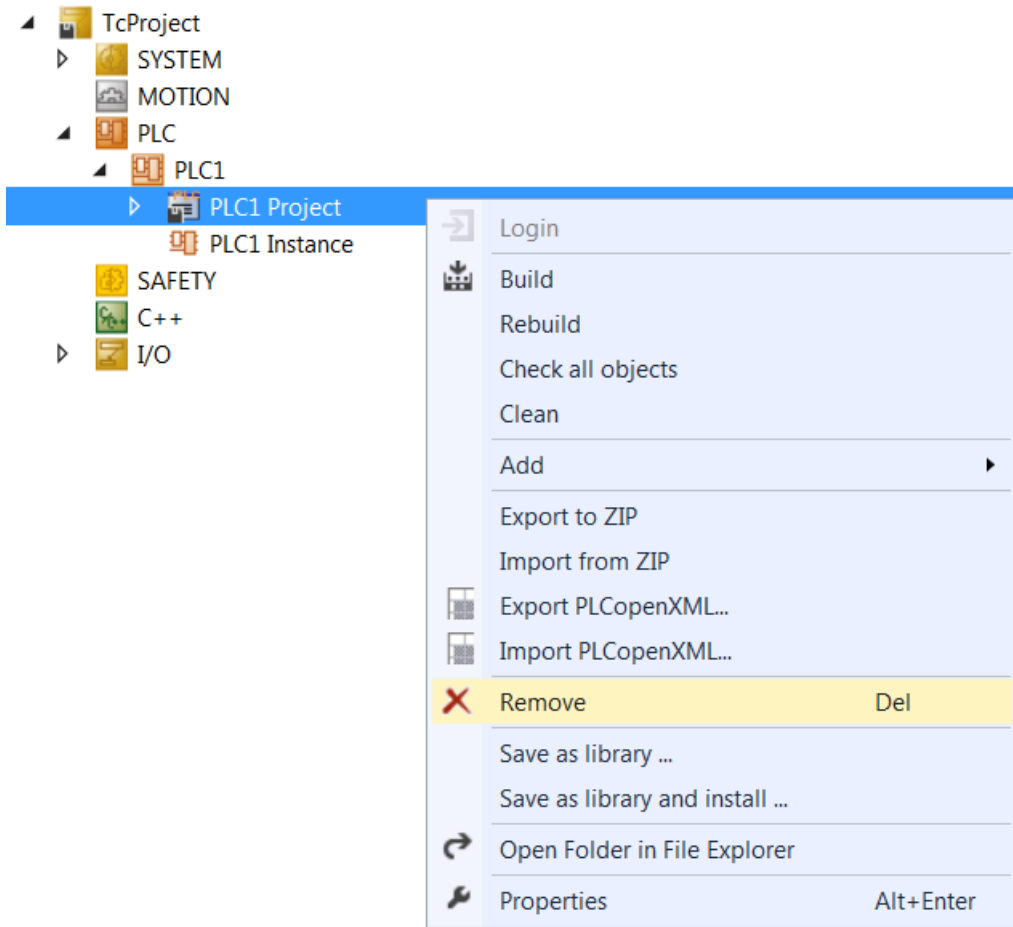


⇒ The **Add Existing Project** dialog opens.

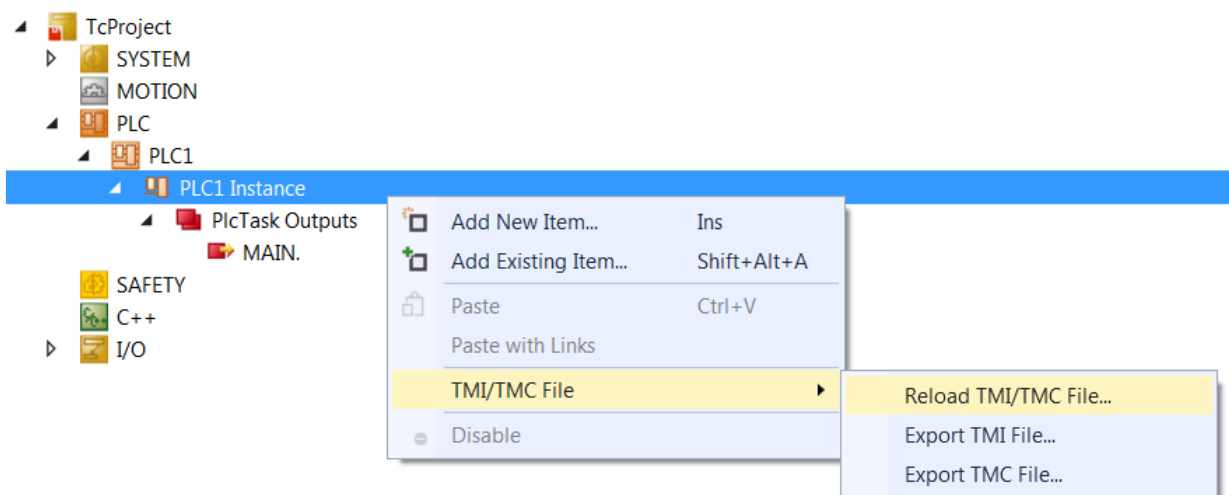


3. Select the option **Copy project to solution folder** to manage the PLC project independently of the original PLC project.
 - ⇒ The PLC project is added to the stand-alone PLC project.
4. Create the PLC project to create a TMC file.

5. Delete the embedded PLC project in the existing TwinCAT project by deleting only the actual PLC project, but not the associated instance, so that the existing mappings are retained.



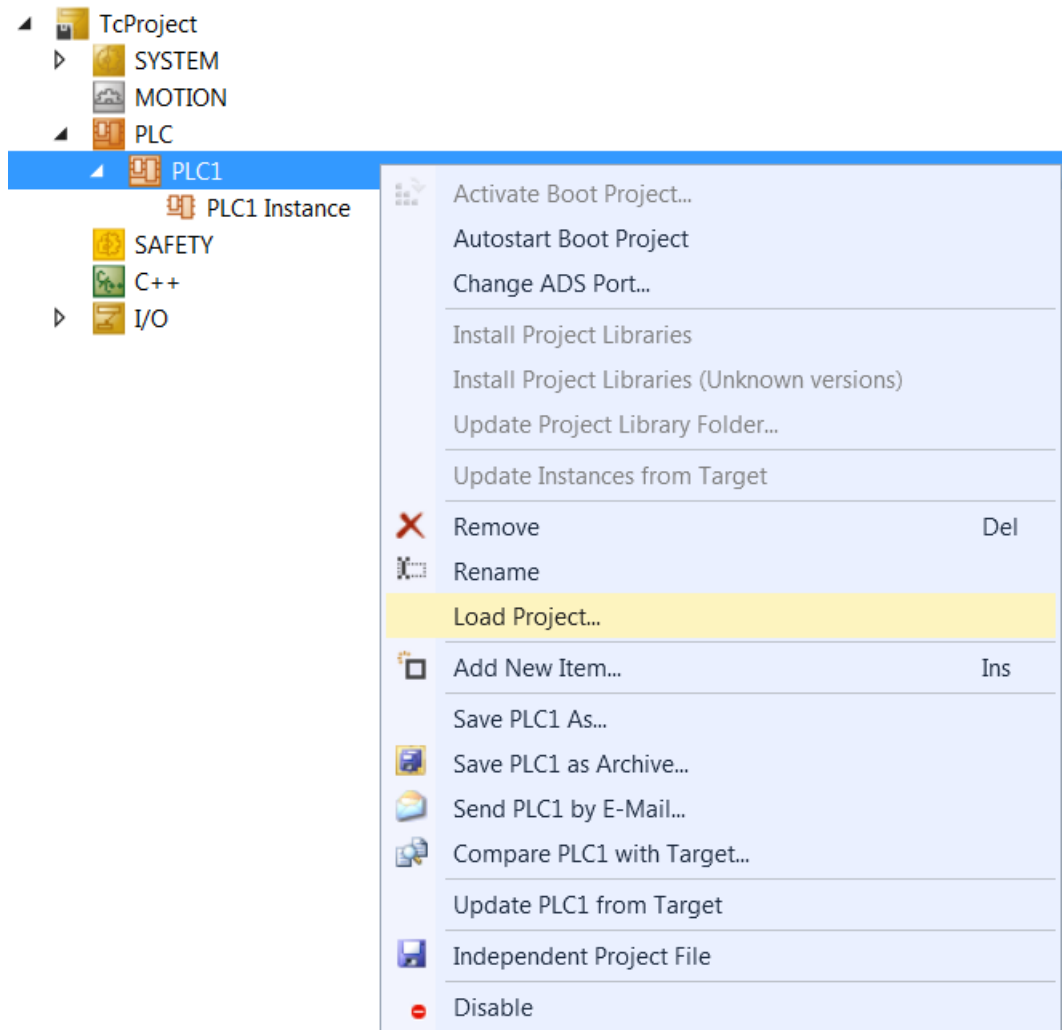
6. Change the path of the TMC file. To do this, click on **TMI/TMC File > Reload TMI/TMC File...** in the context menu of the PLC project instance and select the TMC file of the stand-alone PLC project in the standard dialog that opens. Confirm the dialog.



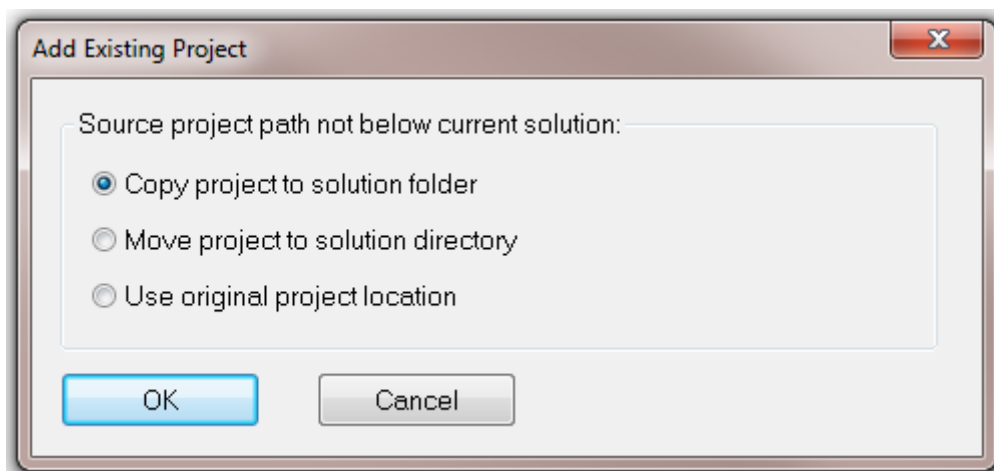
- ⇒ The PLC project previously embedded in the TwinCAT project now forms an independent PLC project (stand-alone PLC project).

Converting a stand-alone PLC project into an embedded PLC project

1. Add the PLC project of the stand-alone PLC project to the existing PLC project. To do this, click on **Load Project...** in the context menu of the PLC project and select the .plcproj file of the stand-alone PLC project in the standard dialog that opens. Confirm the dialog.



⇒ The **Add Existing Project** dialog opens.



- **Copy project to solution folder**, if you want to manage the PLC project independently of the original stand-alone PLC project and continue to use the stand-alone PLC project.
- **Move project to solution directory** if you no longer want to use the stand-alone PLC project.
- **Use original project location** if you want to work with the TwinCAT project and the stand-alone PLC project on the same data.

2. Select the desired option and confirm the dialog.
⇒ The stand-alone PLC project is now embedded in the TwinCAT project.

12.4 FAQ

Overview

- [Why is it not possible to activate the configuration in the TwinCAT project? \[▶ 265\]](#)
- [Why is it not possible to log into the PLC in the stand-alone PLC project? \(a\) \[▶ 265\]](#)
- [Why is it not possible to log into the PLC in the stand-alone PLC project? \(b\) \[▶ 265\]](#)

Why is it not possible to activate the configuration in the TwinCAT project?

Error Message: "Cannot copy file target – source file ‘...' not found"

Cause: This error can have different causes:

1. The ADS port of the stand-alone PLC project and the PLC instance in the TwinCAT project are not identical.
Solution: Set the same ADS port at both levels, recompile the stand-alone PLC project. If the **Auto Reload TMI/TMC** option is not activated in the PLC instance configurator, reload the TMC file.
2. The TwinCAT project and the stand-alone PLC project were not compiled for the same platform.
Solution: Compile both projects for the same platform. If the **Auto Reload TMI/TMC** option is not activated in the PLC instance configurator, reload the TMC file.
3. Only the TMC file was provided for the TwinCAT project, so that the path of the TMC file does not lead to the stand-alone PLC project and the PLC runtime data cannot be accessed.
Solution: Disable the **Activate PLC configuration (copy boot files)** option in the **Settings** tab of the TwinCAT project PLC settings.
In this way, the PLC runtime data can be transferred from the stand-alone PLC project to the target system without exception.

Why is it not possible to log into the PLC in the stand-alone PLC project? (a)

Error Message: "PLC instance parameter (0x0850801a) mismatch. Download will be aborted."

Cause: If this error occurs, the PLC module instance information has not been updated before logging in.

Solution: Acknowledge the error message dialog with **Yes** and ignore the subsequent error messages, or update manually using **Update Instances from Target**.

Why is it not possible to log into the PLC in the stand-alone PLC project? (b)

Error Message: "PLC instance parameter (0x08500005) mismatch. Download will be aborted."

Cause: If this error occurs, the PLC project does not correspond to the PLC instance of the activated TwinCAT project. This means that the TMC file and thus the PLC instance was not updated in the TwinCAT project after changes in the stand-alone PLC project.

Solution: Acknowledge the error message dialog with **Yes** or **No** and ignore the following error messages. Then update the TMC file in the TwinCAT project (**context menu PLC project instance > TMI/TMC File > Reload TMI/TMC File...**). If the TMC file is not provided separately but is added directly via the path of the stand-alone PLC project, the **Auto Reload TMI/TMC** option should be enabled in the PLC instance configurator.

13 Using libraries

Libraries are collections of reusable objects such as:

- POU's such as function blocks or functions
- Interfaces and their methods and properties
- Data types such as enumerations, structures, aliases, unions
- Global variable, constants, parameter lists
- Text lists, image pools, visualizations, visualization elements
- External files (e.g. documentation)

Integration of a library in a project enables the library modules to be used in a project in the same way as the other function blocks and variables, which are defined directly in the project.

Recommendations and notes

In addition to the library use descriptions, see the application notes in [Recommendations and notes](#) [[▶ 268](#)].

The following steps are relevant for the application of libraries. Creation, installation and management of libraries.

If a library was already created and installed (this is the case for system libraries, for example), only the library management or integration step is required. Some libraries have to be created and installed first.

Library creation

- A choice of two library types is available when a library is created: *.library (source library) and *.compiled-library (compiled library with source code protection).
- Prerequisites and further information can be found in section [Library creation](#) [[▶ 269](#)].

Library installation

- Before a library can be used in a project, the library first has to be installed on the system. Libraries are managed on the local system in different "repositories" (directories, storage locations). Before a library can be integrated in a project, it has to be installed in such a repository with a defined version number on the local system.
 - **Exception:** projects referenced as library.
See Use PLC project as referenced library.
- If a library version is used that is not installed in the repository, this is flagged up at the reference in the project tree.
- Libraries are installed in the [Library Repository](#) [[▶ 272](#)].

Library management

- The [Library Manager](#) [[▶ 276](#)] offers a good overview of the PLC library references used in the project and can be used to integrate libraries or placeholders in a project. The integration makes the library reference elements available for use in the project.
- Library references, which are referenced as sub-libraries in another library, are also shown in the Library Manager. In addition, there are "hidden libraries" (see section [Command Properties](#) [[▶ 362](#)]).
- Whenever a library is used, a unique library version is referenced. This version is specified as the effective version. If the library was attached with a fixed version (e.g. 3.3.0.0), the project will always use this library version, even if a more recent version is or becomes available. Alternatively, the setting "Always newest"/"" can be used to automatically ensure that the latest library version is used at all times. In this case, TwinCAT always uses the latest version of the library found in the library repository. For more information and a sample, see the [Command Set to Always Newest Version](#) [[▶ 364](#)].
- It is not possible to add the same version of the same library more than once to a library manager. A version of a library can be referenced in a library manager either as a library or as a placeholder.

- If the library is not compiled (*.compiled-library) but is instead available as a *.library file, the library elements listed in the Library Manager can be opened by double-clicking on the respective entry.
- You can add library references in the form of a library or a placeholder to the Library Manager and include them in your application (see section [Command Add library \[▶ 358\]](#)). Placeholders should be used whenever possible. Further information can be found in section [Library placeholders \[▶ 279\]](#).
- When a library module is addressed in the project, the libraries and repositories are searched in the order in which they are listed in the [Library Repository \[▶ 272\]](#). Further information can be found in section [Unambiguous access to library modules or variables \[▶ 267\]](#).

Library documentation

TwinCAT offers a wide range of library documentation options. Further information can be found in section [Library documentation \[▶ 282\]](#).

Refer also to the following for information on these topics:

- Referenced libraries
- Library versions
- Unambiguous access to library modules and variables
- TwinCAT 2.x PLC Control libraries
- External and internal libraries or library modules, late binding

Referenced libraries

- A library can integrate other libraries (referenced libraries), whereby the nesting can be as deep as required. If such a "father" library is then itself integrated in a project, the libraries referenced in it are available there too.
- Library references should always be defined via [library placeholders \[▶ 279\]](#), in order to avoid problems that may arise through version dependencies or the need to use manufacturer-specific libraries.
- In the [properties \[▶ 362\]](#) for each referenced library you can specify how it should behave later, when it is integrated in a project via the "father" library, e.g. whether it should be "hidden" in the Library Manager.

Library versions

- Several versions of a library can be installed on the system at the same time.
- Several versions of a library can be integrated in the project at the same time. However, this is not advisable. In this case, a unique namespace must be assigned to each of the libraries, and access to the symbols must be qualified. Examples: V1.Send, V2.Send
- The version or resolution of libraries or placeholders can be configured in the [Properties window \[▶ 362\]](#). The resolution of placeholders can also be adapted in the Placeholder dialog.
- It is strongly recommended to use placeholders, if other libraries are referenced in a library, but also in order to make a project compatible. In this way it is possible to avoid problems arising from version dependencies or the need to use manufacturer-specific libraries.

Unambiguous access to library modules or variables

- If several modules or variables with the same name are available in the project and in libraries, access to a module component should be unambiguous. Related to libraries, uniqueness is achieved by adding the namespace of the library before the module name.
- The default setting for the **namespace** of a library is the library title. On the other hand, a different namespace can also be explicitly defined for a library: either generally for the library when [creating the library \[▶ 269\]](#) in the [project properties \[▶ 906\]](#) or for local use of the library in a project in the [properties](#)

[window](#) [[▶ 362](#)] of the library reference.

The namespace of the library must be used as a prefix of the identifier so that unique access is possible to a module that exists multiple times in the project.

- Sample:
 - The library Lib1 is integrated in an application project.
 - The function F_Sample is declared both in library Lib1 and in the application.
 - In order to implement access to the two functions, the namespace of the library is added before the library function is called. This makes reference to the function of library Lib1 unambiguous.
 - Calling the application function `nResult := F_Sample(nInput := nVar);`
 - Calling the library function `nResult := Lib1.F_Sample(nInput := nVar);`

TwinCAT 2.x PLC Control libraries

- Libraries that were created with TwinCAT 2.x PLC Control (*.lib) continue to be supported.
- An "old" library project (*.lib) can be used in TwinCAT 3 PLC and converted to a "new library" (*.library/*.compiled-library).
- If an existing project that references old libraries is opened, the user can select whether these references should be retained, replaced by others, or deleted. If they are to be retained, the respective libraries are converted to the new format and automatically installed in the "System" library repository. If they do not contain the required project information, it can be added directly. The procedure, based on which a particular old library was handled during conversion of an old project, can be stored in the project options. If the same library then reappears in the conversion of another old project, the procedure does not have to be defined again, but is executed automatically.
- Section [Add New Item...](#) [[▶ 896](#)] describes the procedure for opening and converting projects and libraries.

External and internal library modules, late binding

- External library modules are firmware functions whose implementation is not included in the PLC library. For libraries with firmware functions, the firmware must be available on the target system; it is not bound until the application runs there.

13.1 Recommendations and notes

Some recommendations and notes relating to library usage are described below.

Recommendation of library placeholders

Through the use of library placeholders the adaptation of the library versions used requires very little effort, thus making the process of engineering projects and libraries very flexible. We therefore recommended using placeholders instead of libraries.

For more information, see [Library placeholders](#) [[▶ 279](#)].

Possible online change through the use of "always newest" versions

Sample: You activate and start a project in which the library "LibA" is referenced as "always latest" version. At the time of the download, version 1.0.0.0 is the latest version of this library in the library repository. If you then install a newer version of "LibA" in the library repository (e.g. version 1.0.1.0), this version becomes the latest version of the library. If you log in with the application project, which you have not actively changed, TwinCAT detects a change in the application because "LibA", which is used as "always latest", is no longer referenced in version 1.0.0.0, but now in version 1.0.1.0. This means that you will be offered an online change or download when logging in, although you may not be aware of any project changes.

This irritation due to automatic changing of the effective version of a library (caused by the "always latest version" setting) is unnecessary and should be avoided. In addition, the project sources and the components used should be frozen and backed up for tracking purposes after commissioning or after completion of the project.

Therefore, we strongly recommended that all library references (i.e. library references directly at project level and library references used internally in libraries) be set to a fixed effective version after project completion (i.e. before delivery).

Note that this recommendation applies not only to user libraries but also to Beckhoff libraries, as new versions of referenced Beckhoff libraries may be installed with the installation of a new XAE or RM setup.

During the development phase, however, it makes sense to use the "always latest version" option, as you will then always automatically work with the latest available library versions.

For more information about how to set library versions to a fixed effective version, see section [Command Set to Effective Version](#) [▶ 364].

For more information and another example of the behavior of "always newest" versions, see the [Command Set to Always Newest Version](#) [▶ 364].

Transferring source code of user libraries

If you have configured the target or file/email archive settings to include source libraries in one of these archives, please note that the source libraries (*.library) used in the project are contained in the ZIP archive in readable source code form when passing on/delivering the target system or when passing on the file/email archive. Keep this in mind when referencing libraries and when passing on the target system or archives.

If you do not want to share the readable source code of (user) libraries in the target or file/email archive, you can either save and reference these libraries as compiled libraries (*.compiled-library) or you can disable adding source libraries to the target or file/email archive in the project settings.

For more information on the configuration options of the target and file/email archives, see section [PLC Project Settings > Settings tab](#) [▶ 926].

Information about source code encryption can be found in the documentation on [Security Management](#).

13.2 Library creation

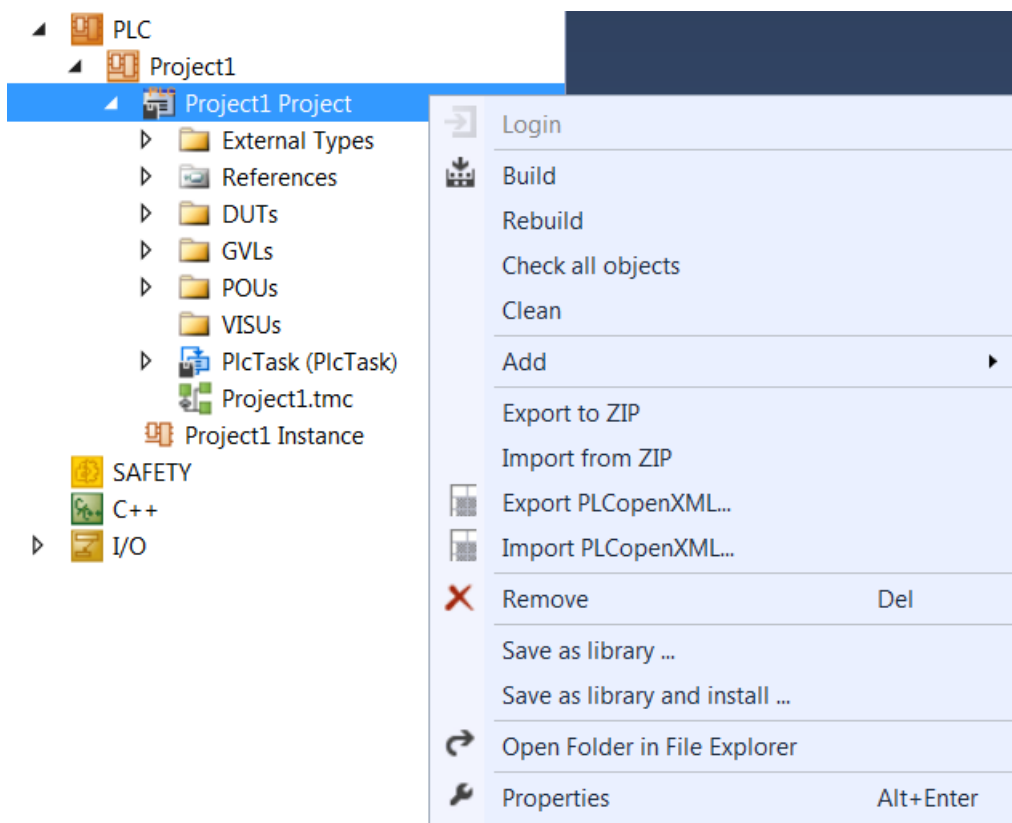
- A TwinCAT 3 PLC project can be stored as a library (<Project name>.library/.compiled-library). At the same time it can be installed in the [Library Repository](#) [▶ 272], if required. To create a specific library project, it is recommended to select the template **Empty PLC project** directly in the **New Project** dialog.
- In order to be able to save a project as a library, a **title**, a **version number** and the **company name** must be entered in the [project properties](#) [▶ 906].
- If the library integrates other libraries, the user should carefully consider how these referenced libraries should behave later, when the "father" library is integrated in a project. These considerations apply to version handling, namespace, visibility and access options, some of which can be configured in the Properties dialog of the individual referenced libraries. A placeholder can be used when the reference is defined, if the library is to always reference another library when it is integrated in a project.
- To make library modules view- and access-protected, a library project can be saved in a precompiled format (<project name>.compiled-library) – see [Command Save as library](#) [▶ 270].
- Data structures of a library can be marked as library-internal. These non-public objects have the access label INTERNAL or the [attribute 'hide'](#) [▶ 805] and therefore do not appear within the [Library Manager](#) [▶ 276], the "List components" function or the input wizard.
- A comment can be added to the declaration of a function block parameter, as a convenient way to make library function block information available to the library user. This comment is later shown on the **Documentation** tab in the [Library Manager](#) [▶ 276], when the library is integrated in a project. Also note the [Library documentation](#) [▶ 282] options.
- The following commands for saving a library are available in the context menu of the PLC project:
 - [Command Save as library](#) [▶ 270]
 - [Command Save as library and install](#) [▶ 271]

13.2.1 Command Save as library

Function: The command opens the standard dialog for saving a PLC project as a PLC library.

Call: Context menu of the PLC project object (<PLC project name> Project) in the Solution Explorer

A PLC project can be saved as a PLC library in order to make source code available for other applications as a library and therefore via a defined interface. The command for saving a library is available in the context menu of the PLC project.

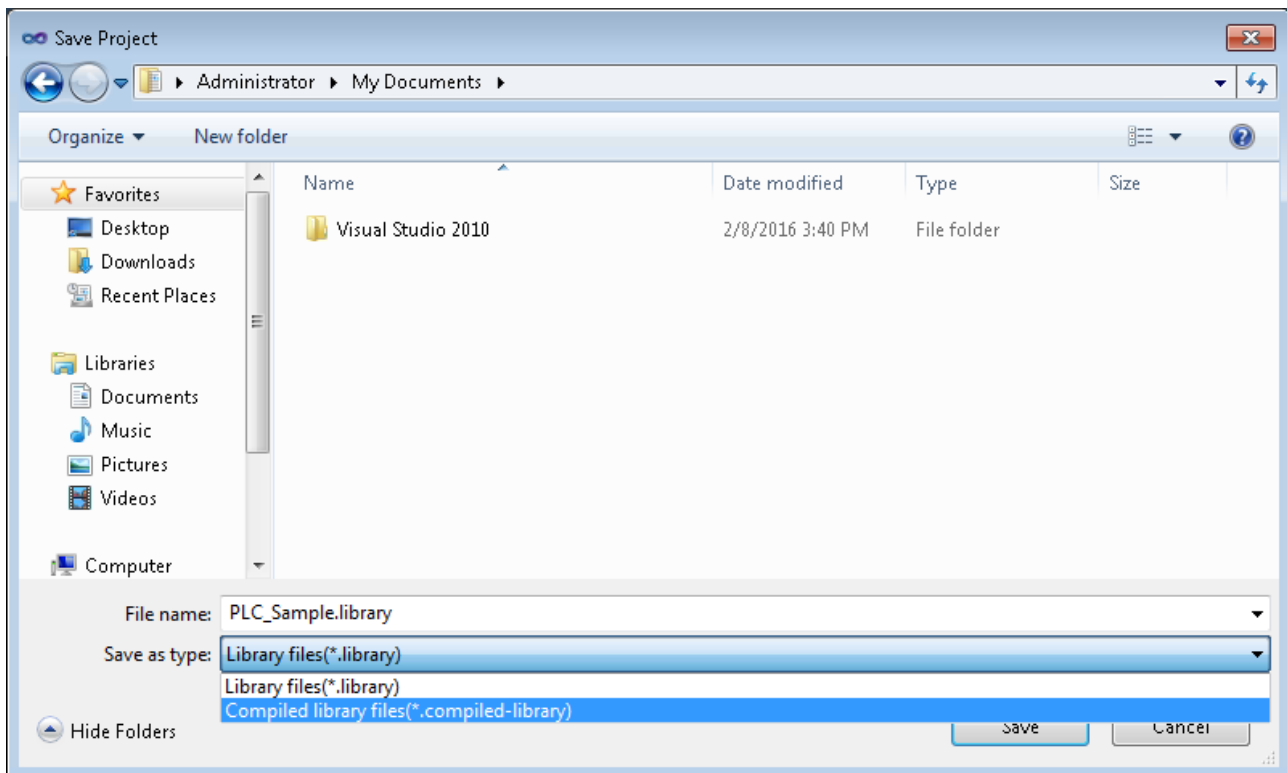


The command opens the standard dialog for saving a file in the file system. The existing project name is offered automatically; it can be modified by the user, if required. When a project is saved as a library, there is a choice between two library file formats:

- *.library (source library)
 - You can open a source library (for viewing and/or editing) by using the **Add Existing Item** command, which is available on the PLC node within the project tree.
 - You can "step" into a source library using the usual debugging function.
- *.compiled-library (compiled library)
 - This file extension can be used to save a library project in a compiled format. An encrypted image of the precompile context of the library is stored, which means the implementations of the library function blocks are no longer accessible or visible.
 - A compiled library cannot be opened or debugged.
 - Apart from this, *.compiled-library files behave like *.library files. You can therefore install and reference them in the same way.
 - The source code of a library can be protected by using the compiled library format. In addition, the library files are smaller and the loading times are shorter.

i The code cannot be run through step by step

The usual debugging functions cannot be applied to a compiled library (*.compiled-library). It is therefore not possible to jump into a library block of a *.compiled-library via debugging.

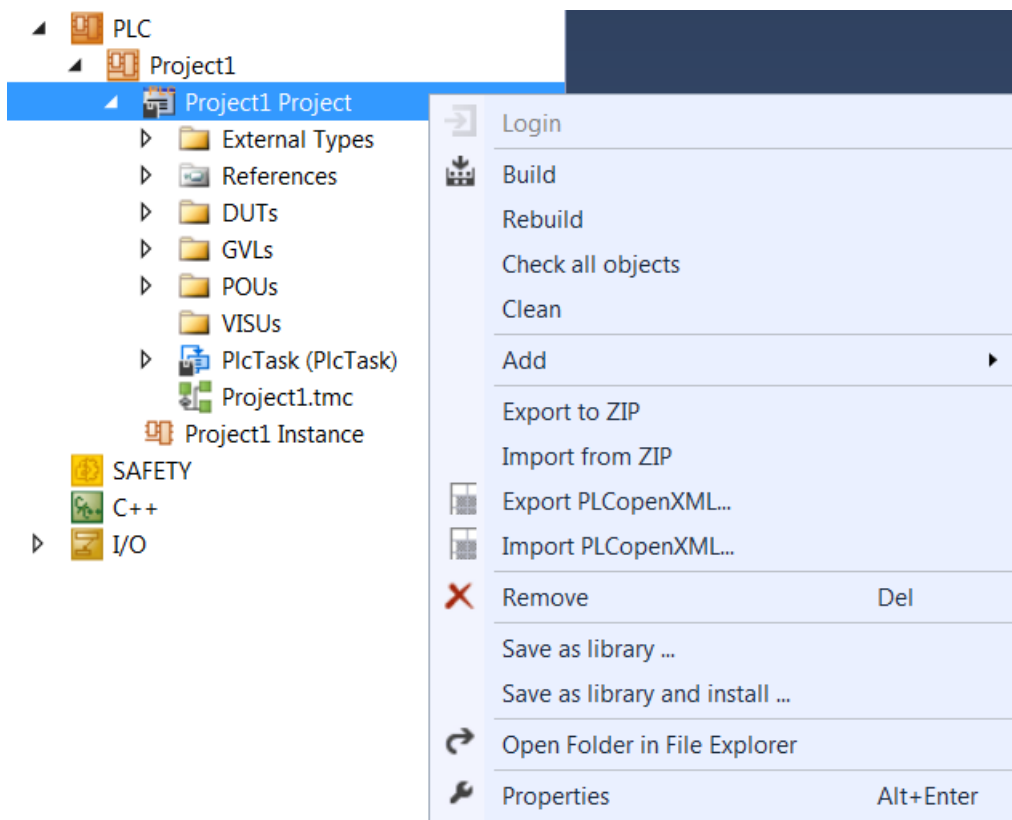


13.2.2 Command Save as library and install

Function: The command opens the standard dialog for saving a PLC project as a PLC library. In addition, the command installs the stored library in the library repository. The library can thus be inserted directly into a project via the Library Manager.

Call: Context menu of the PLC project object (<PLC project name>project) in the Solution Explorer

This command saves the PLC project as a PLC library and installs it in the [Library Repository](#) [▶ 272]. The command for saving and installing a library is available in the context menu of the PLC project.



The installation of the library, which is performed in addition to the saving, is an extension of the [Command Save as library](#) [▶ 270], since the library is installed on the local system at the same time. The library is then immediately available for addition to a project via the Library Manager.


13.3 Library installation

- Before a library can be used in a project, the library first has to be installed on the system. Libraries are managed on the local system in different "repositories" (directories, storage locations). Before a library can be integrated in a project, it has to be installed in such a repository with a defined version number on the local system.
 - **Exception:** projects referenced as library.
See [Use PLC project as referenced library](#).
- If a library version is used that is not installed in the repository, this is flagged up at the reference in the project tree.
- Libraries are installed in the [Library Repository](#) [▶ 272].

13.3.1 Library Repository

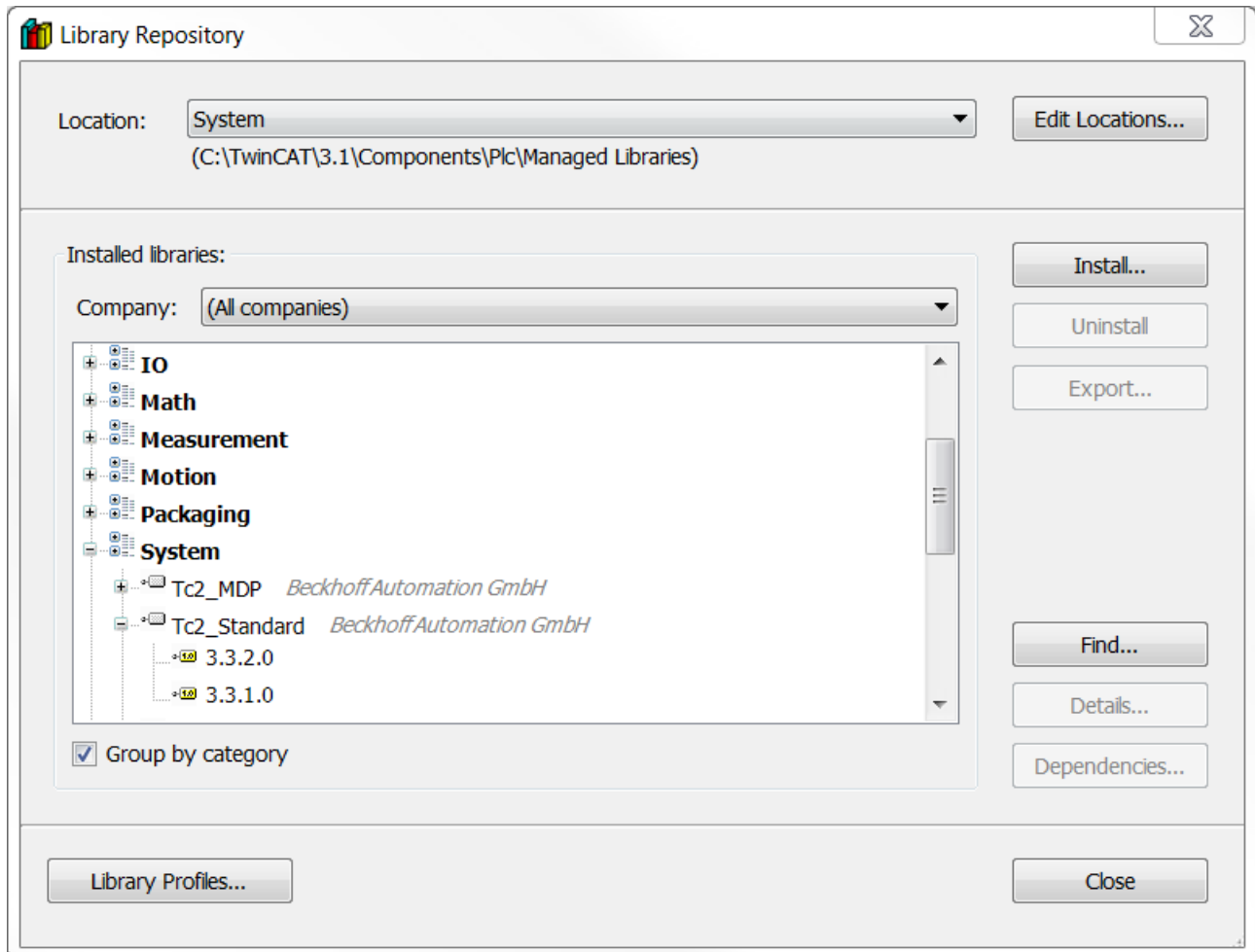
Function: The Library Repository can be used for the definition of storage locations and for the installation or uninstallation of libraries.

Call:

- Menu **PLC**
- Context menu of the **References** object in the PLC project tree
- Button in [Library Manager](#) [▶ 276] (symbol: )
- Via the extended dialog **Find library** after using the command [Add library without placeholder resolution command](#) [▶ 359]

In order to be able to use a library, it must be installed in the repository. For the Beckhoff libraries this generally happens during the TwinCAT 3 installation or during the installation of TwinCAT 3 functions. Libraries can be installed as source library (*.library) or as compiled library (*.compiled-library). Beckhoff supplies compiled libraries. If an attempt is made to use a library version that is not installed in the repository, a note symbol appears at the reference in the project tree.

The library repository contains all installed libraries. This list can be sorted and displayed based on the library categories (option **Group by category** is enabled) or alphabetically based on the library titles (option **Group by category** is disabled). When the libraries are sorted based on categories, the categories appear as nodes. Clicking on a node opens the list of associated libraries or subcategories; clicking on a library name opens the list of installed library versions.



Buttons and commands

The following buttons and commands are available in the **Library Repository**.

Location

Displays the directories on the local system, in which the library files are stored. The libraries for this storage location are listed in the **Installed libraries** section. If several repository directories exist on the system, a repository directory can be selected for management at this point.

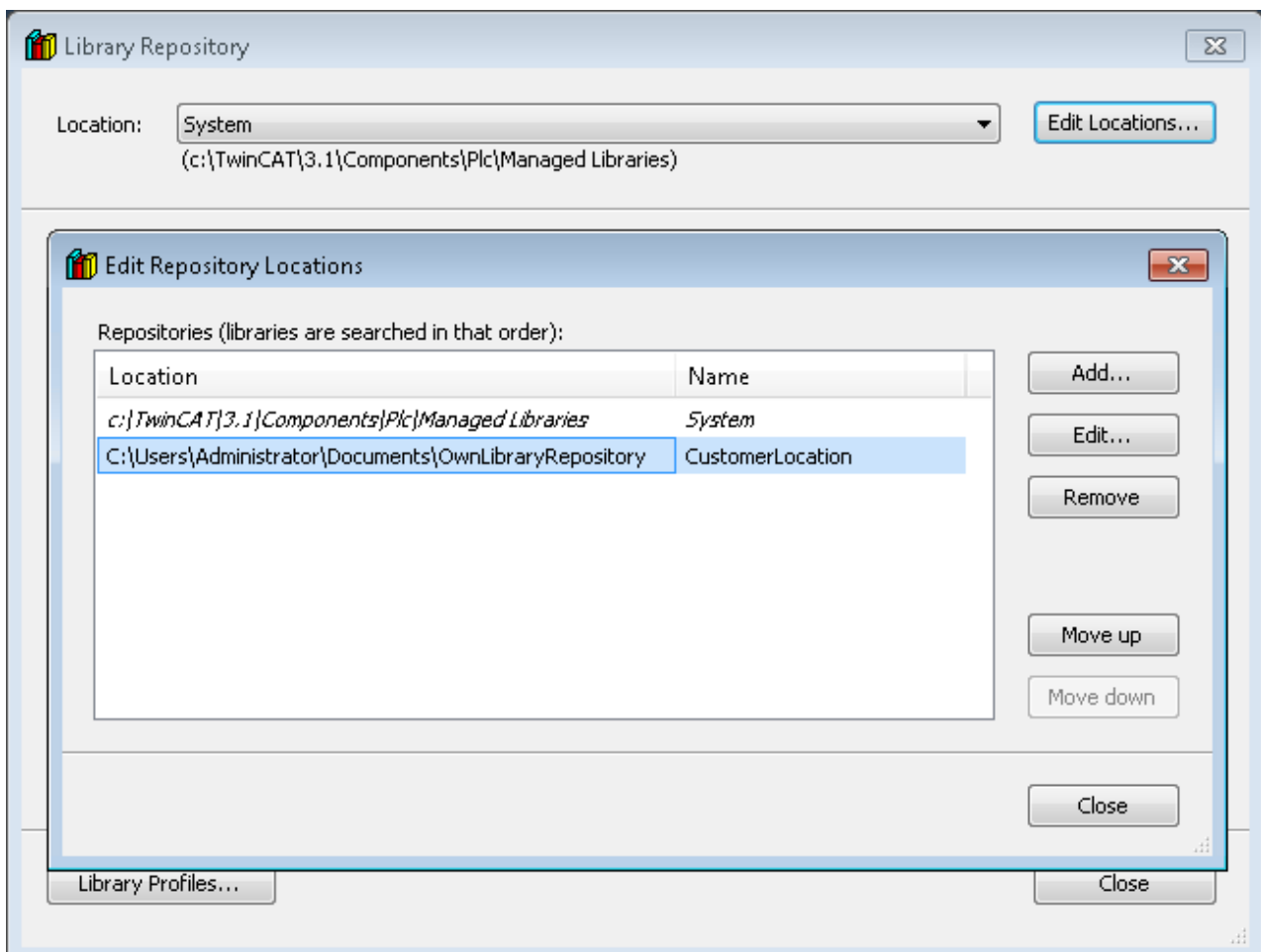
Edit	Opens the dialog Edit Repository Locations .
------	---



For new repositories you can only use empty directories or a valid existing repository.

Edit Repository Locations

List the repositories with location and name.	
Add	Creates a new repository by specifying a repository name and directory. Opens the dialog Repository Location . The selected directory (Location input field) must be empty or an existing valid repository. Name is the input field for a symbolic repository name.
Edit	Opens the Repository Location dialog (see "Add") for editing the currently select library repository.
Remove	The user is asked whether only the repository list entry is to be removed, or whether the whole directory containing the library files should be deleted from the file system. If you want to delete the directory, you need to confirm this.
Move up	Command for changing the repository order, by moving the currently selected repository up by one position.
Move down	Command for changing the repository order, by moving the currently selected repository down by one position.



Installed libraries

List of libraries in a tree structure. Each library is shown with category, name, company and version.	
Company	Selection list for filtering the displayed libraries.
Install	Opens the Select Library dialog for selecting a library file in the file system. The selected library file of type *.library or *.compiled-library is installed in the local Library Repository. The library can then be used in projects.
Uninstall	Uninstalls the selected library version. It can then no longer be used in projects.
Export	This command allows you to export a library that is installed in your library repository and store it at the desired location. The command opens the Export Library dialog for selecting a location. The library can be exported in the format in which it was installed in the repository (as source library *.library or as *.compiled-library).
Search	Opens the Find Library dialog for finding library names or library elements. Double-click on a search result or select a search result + [Open] to close the search dialog and the corresponding library is marked in the Library Repository.
Details	Opens the Details [▶ 361] dialog for the selected version of a library.
Dependencies	Opens the Dependencies [▶ 362] dialog for the selected version of a library.
Group by category	<ul style="list-style-type: none"> • <input checked="" type="checkbox"/> : Grouping by library categories • <input type="checkbox"/> : Alphabetical sorting <p>The categories are defined through external description files “*.libcat.xml”.</p>

Library profiles

A library profile defines with which library version TwinCAT resolves a library placeholder, if a certain compiler version is set in the project.	
Import	Imports a *.libraryprofile file. If the import contains existing placeholder entries, a prompt appears asking whether TwinCAT should overwrite them.
Export	Exports an xml file with the extension “.libraryprofile”, which contains the assignments of the selected placeholder entries. You can limit the selection to a single entry from a compiler version.

Converted TwinCAT 2 libraries in the Library Repository

The Library Repository contains converted TwinCAT 2 libraries (Tc2_<LibraryName>) and new TwinCAT 3 libraries (Tc3_<LibraryName>), which are not available in TwinCAT 2.

These libraries enable conversion of a TwinCAT 2 project, which uses TwinCAT 2 libraries, to a TwinCAT 3 project, which uses compatible TwinCAT 3 libraries, without the need for major changes in the PLC code. The Tc2_<LibraryName> libraries are therefore converted TC2 libraries. The naming of the libraries in TwinCAT 3 is similar to that in TwinCAT 2. To simplify matters, some libraries were combined. The project converter automatically assigns the TC2 libraries of a TwinCAT 2 .pro file to the TC3 libraries in a TwinCAT 3 project.

A list of PLC library assignments TC2 library → TC3 library is stored in the TwinCAT options and can be expanded there:

Tools\Options\TwinCAT\PLC Environment\Libraries

Older libraries must be replaced before a project is converted (i.e. MC → MC2, COMlib → COMlibv2, PLCSYSTEM → TcSystem, ...).

13.4 Library management

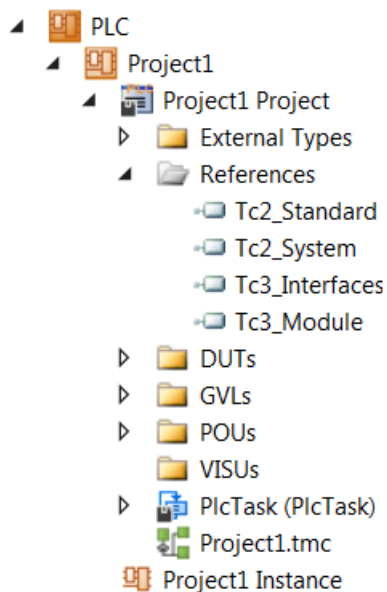
- The [Library Manager \[▶ 276\]](#) offers a good overview of the PLC library references used in the project and can be used to integrate libraries or placeholders in a project. The integration makes the library reference elements available for use in the project.

- Library references, which are referenced as sub-libraries in another library, are also shown in the Library Manager. In addition, there are “hidden libraries” (see section [Command Properties](#) [▶ 362]).
- Whenever a library is used, a unique library version is referenced. This version is specified as the effective version. If the library was attached with a fixed version (e.g. 3.3.0.0), the project will always use this library version, even if a more recent version is or becomes available. Alternatively, the setting "Always newest"/"" can be used to automatically ensure that the latest library version is used at all times. In this case, TwinCAT always uses the latest version of the library found in the library repository. For more information and a sample, see the [Command Set to Always Newest Version](#) [▶ 364].
- It is not possible to add the same version of the same library more than once to a library manager. A version of a library can be referenced in a library manager either as a library or as a placeholder.
- If the library is not compiled (*.compiled-library) but is instead available as a *.library file, the library elements listed in the Library Manager can be opened by double-clicking on the respective entry.
- You can add library references in the form of a library or a placeholder to the Library Manager and include them in your application (see section [Command Add library](#) [▶ 358]). Placeholders should be used whenever possible. Further information can be found in section [Library placeholders](#) [▶ 279].
- When a library module is addressed in the project, the libraries and repositories are searched in the order in which they are listed in the [Library Repository](#) [▶ 272]. Further information can be found in section [Unambiguous access to library modules or variables](#) [▶ 267].

13.4.1 Library Manager

Function: The Library Manager is used to integrate and manage libraries in a project. It provides a good overview of the PLC libraries used in the project.




Call: Double-click on the **References** object in the PLC project tree



Compilation errors relating to the Library Manager are output in a dedicated category in the message window.

Upper part of the Library Manager

The upper part of the dialog shows the libraries currently integrated in the project.	
Name	Title, version and company name, as defined in the project properties [▶ 906] during library creation [▶ 269] .
Namespace	The default setting for the namespace of a library is the library title. Alternatively, a different namespace can be defined explicitly, either generally for the library in the project properties [▶ 906] during library generation [▶ 269] , or in the Properties window of the library reference [▶ 362] for local use of the library in a project. The namespace of the library must be used as prefix of the identifier, in order to enable unambiguous access to a module that is used several times in a project.
Effective version	Effective library version, which is referenced. If the library is used as "Always newest"/"", the actual library version is displayed here. Further information on the resolution of placeholders can be found in section Placeholder [▶ 280] .

- Libraries that were automatically inserted into the project by a plug-in (e.g. visualization or UML libraries) are displayed in gray font. Manually added libraries (**Add library...**) are shown in black font.
- A symbol before the library name indicates the library type:
 -  : TwinCAT 3 PLC library (contains version information)
 -  : The referenced file was not found or is not a valid library file (see corresponding message in the **Error List** view). In this case see: [Command Try reload library \[▶ 360\]](#)
- If a library has [dependencies \[▶ 362\]](#) from other libraries, these are – if they are found – also integrated automatically and shown in a subtree of the entry, preceded by a symbol . Such a subtree can be opened or closed via a plus or minus sign. As an example, the diagram below shows the library "Tc2_Standard" as a direct library and as a sub-library of the library "Tc2_System".

Buttons and commands

The following buttons and commands are available in the upper section of the Library Manager.

Add library	Integrating a library or a placeholder in the project. See also Command Add library [▶ 358] .
Delete library	The command removes the selected library from the Library Manager.
Details	Opens the Details [▶ 361] dialog for the selected library.
Placeholder	Opens the Placeholder [▶ 280] dialog.
Library Repository	Opens the Library Repository [▶ 272] dialog.
Try reload library	This command is available in the editor window of the Library Manager, if a library is selected which failed to load when the project was opened. See also Command Try reload library [▶ 360]

Lower part of the Library Manager

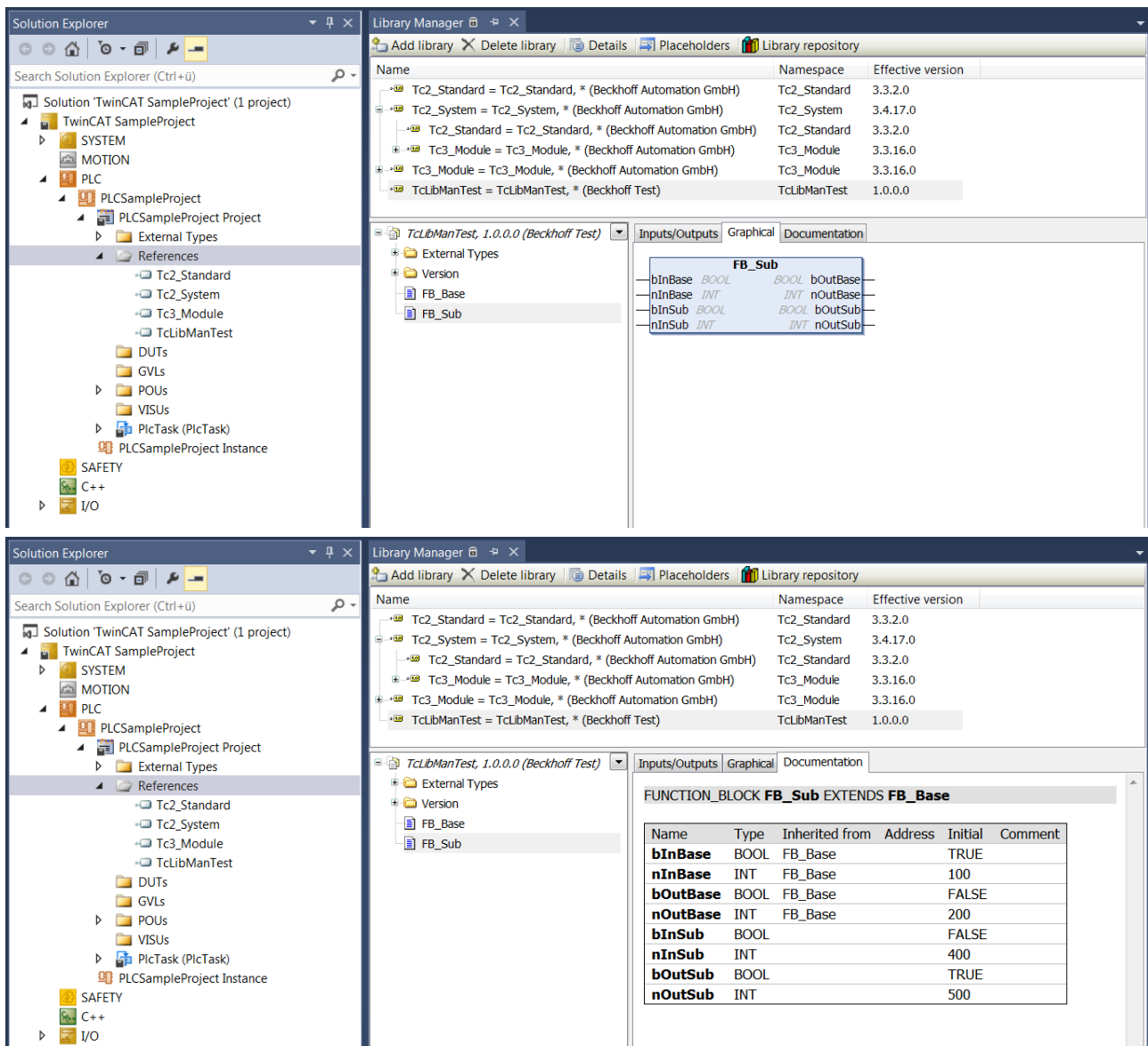
The lower left part of the editor shows the modules of the library that is selected in the upper part of the editor. The usual sorting and search functions are available via a menu symbol shown to the right of the library name.

The following tabs can be found in the lower right section.

Inputs/Outputs	<p>The components of the library object currently selected on the left are shown in a table, with (variable) name, data type, basic function block if applicable, address, basic function block if applicable, initial value and comment, as defined in the library.</p> <p>The symbol in front of each variable indicates whether it is an input, output or input/output variable.</p> <ul style="list-style-type: none"> • Input variable: Symbol with arrow pointing down and right • Output variable: Symbol with arrow pointing up and left • Input/output variable: Symbol with arrow pointing down/right and up/left <p>The symbol for method return values is the same as the symbol for output variables.</p>
Library parameters	<p>This tab is only available if the library object currently selected on the left is a parameter list.</p> <p>The variables of the parameter list are displayed in a table with name, data type, (editable) value and comment, as defined in the library.</p> <p>In the (editable) Value column, you can replace the value of the global constant with a project-specific value. For more information, see Object Parameter List [▶ 73].</p>
Graphical	Graphical representation of the function block
Documentation	<p>The components of the library object currently selected on the left are shown in a table with (variable) name, data type, basic function block if applicable, address, initial value and a comment, which may have been added to the declaration when the library was created [▶ 269]. Adding this kind of comments is therefore a convenient way to automatically make function block documentation available to the user. Further information can be found in section Library documentation [▶ 282].</p>

The screenshot displays the TwinCAT Library Manager interface. On the left, the Solution Explorer shows the project structure, including the 'References' section with libraries like 'Tc2_Standard', 'Tc2_System', 'Tc3_Module', and 'TcLibManTest'. The main window shows the 'Library Manager' with a list of libraries and their details. The 'TcLibManTest' library is selected, and the 'Inputs/Outputs' tab is active, displaying a table of variables.

Name	Type	Inherited from	Address	Initial	Comment
FUNCTION_BLOCK FB_Sub EXTENDS FB_Base					
bInBase	BOOL	FB_Base		TRUE	
nInBase	INT	FB_Base		100	
bOutBase	BOOL	FB_Base		FALSE	
nOutBase	INT	FB_Base		200	
bInSub	BOOL			FALSE	
nInSub	INT			400	
bOutSub	BOOL			TRUE	
nOutSub	INT			500	



13.5 Library placeholders

A library placeholder is a placeholder that references a particular library. The placeholder can either be resolved to a fixed version or to the "always latest" version of this library.

The instructions on how to create a new library placeholder can be found in the chapter [Add library command](#).

The resolution of all the placeholders in the project is set at the application level. This means that, at the application level, you can set the resolution of the placeholders that are directly integrated at the application level as well as those that are used within referenced libraries.

In other words, you can specify from the outside to which library a library placeholder that is integrated within another library should be resolved. It is not necessary to save the outer library again to change the versions of the internally used library placeholders. For more information on how to set the resolution of placeholders, see section [Changing the placeholder resolution](#) [▶ 281].

On account of these options the adaptation of the library versions used requires very little effort, thus making the engineering process of projects and libraries very flexible.

When a library reference is integrated in a project, it is advisable to use a placeholder instead of a library.

Example

- An application project requires the libraries LibA and LibB.

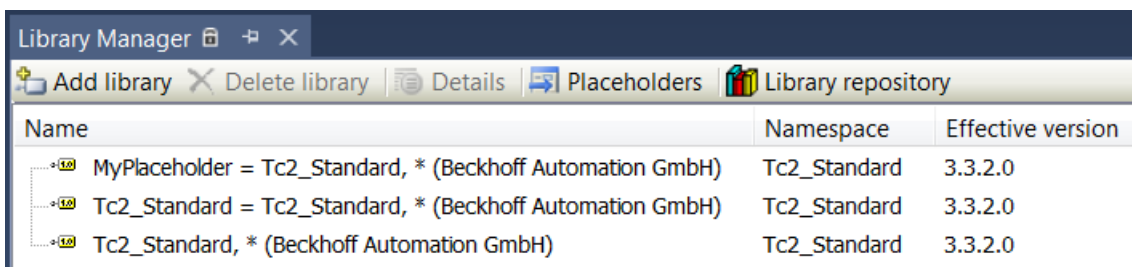
- The library LibB also requires access to the modules of LibA.
- LibA is therefore used as direct library in the application, but also as sub-library of LibB.
- Through referencing of LibA as a placeholder (both in the application and in LibB), definition of the placeholder resolution in the application project enables the user to specify which version of LibA should be used in the whole project (application and LibB).
- If, for example, a change is required within LibA, while the interfaces of LibA remain unchanged and compatibility with the previous version is ensured, this makes it easy to use the new library version in the whole project (application and LibB).
To this end, the modified LibA is created and installed with a new version number (e.g. change from version 1.0.0.0 to 1.0.0.1). If the placeholder LibA was resolved in the application project as a fixed library version, i.e. 1.0.0.0, the placeholder resolution is changed accordingly, to LibA version 1.0.0.1. If, however, the placeholder LibA was defined as “latest version” in the application project, no adaptation is required, i.e. after installation of LibA version 1.0.0.1 this is automatically referenced and used in the whole project (application and LibB).

Detecting libraries and placeholders

Whether a library reference is integrated as a library or as a placeholder can be determined based on differences in the [Library Manager](#) [▶ 276] and in the [Properties window](#) [▶ 362].

- Library Manager: if a placeholder is used, the library name is assigned to the placeholder name, to indicate which "actual" library the placeholder refers to (e.g. MyPlaceholder = Tc2_Standard, * (Beckhoff Automation GmbH)). This is not the case for a library: only the library name is displayed here (Tc2_Standard, * (Beckhoff Automation GmbH)).

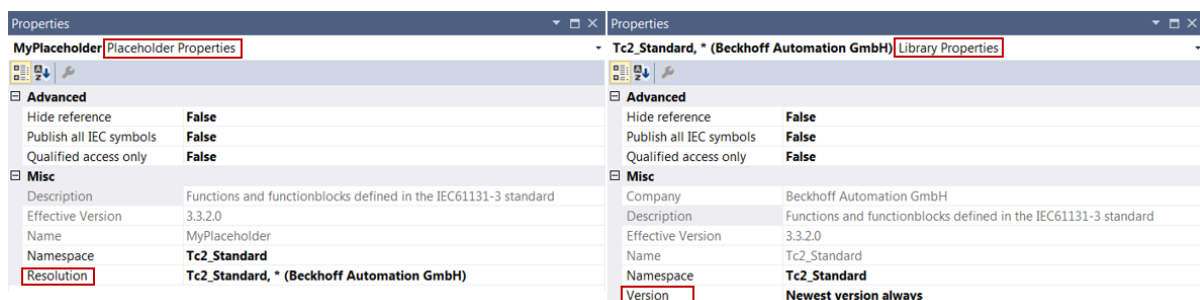
In the following screenshot, the first two entries in the library manager are placeholders, the third entry signals the use of a library.



Name	Namespace	Effective version
MyPlaceholder = Tc2_Standard, * (Beckhoff Automation GmbH)	Tc2_Standard	3.3.2.0
Tc2_Standard = Tc2_Standard, * (Beckhoff Automation GmbH)	Tc2_Standard	3.3.2.0
Tc2_Standard, * (Beckhoff Automation GmbH)	Tc2_Standard	3.3.2.0

- Properties window: the first row of the Properties window indicates whether the element refers to placeholder or library properties.

The properties displayed in the Properties window differ accordingly: a placeholder is **resolved** to a particular library including a particular library version, whereas a library is fixed, and only the **version** can be adjusted. The namespace can be configured in the Properties window in both cases.




Placeholder Properties		Library Properties	
Advanced		Advanced	
Hide reference	False	Hide reference	False
Publish all IEC symbols	False	Publish all IEC symbols	False
Qualified access only	False	Qualified access only	False
Misc		Misc	
Description	Functions and functionblocks defined in the IEC61131-3 standard	Company	Beckhoff Automation GmbH
Effective Version	3.3.2.0	Description	Functions and functionblocks defined in the IEC61131-3 standard
Name	MyPlaceholder	Effective Version	3.3.2.0
Namespace	Tc2_Standard	Name	Tc2_Standard
Resolution	Tc2_Standard, * (Beckhoff Automation GmbH)	Namespace	Tc2_Standard
		Version	Newest version always

13.5.1 Placeholder

Function: The command opens the **Placeholder** dialog. The placeholder overview lists all library placeholders in the project and their current resolution definition. This dialog can also be used to manage placeholders and specify placeholder library versions (e.g. resolution to a particular library version or “Always newest”/“*”). Version resolution at this point is also possible for placeholders that are only used internally in other libraries.

Call:

- Context menu of the **References** object in the PLC project tree
- Button in the [Library Manager \[▶ 276\]](#) (symbol: )

Placeholders are resolved by default to the "Always newest"/"" version of a library. In the placeholder dialog you can redirect every placeholder resolution that is used directly or internally in the project to a different version of the library or resolve the placeholder to a different library.

Structure of the dialog

The **Placeholder** dialog is divided into three columns.

Name	Name of the placeholder
Library	<p>Name of the library to which the placeholder is resolved</p> <ul style="list-style-type: none"> • Bold marking of the library column: The library is marked in bold in the Placeholder dialog if the placeholder is not resolved according to the default resolution (usually "Always newest"/""). This is the case when it has been changed to a user-specific resolution. In this case the Info column indicates "Resolved by placeholder redirection". • Changing the library to which the placeholder is to be resolved: Double-clicking on a field in the Library column opens a selection list below the selected row, which you can use to change the library to which the placeholder is to be resolved. Further information can be found under Changing the placeholder resolution [▶ 281].
Info	<p>Information about the type of placeholder resolution</p> <ul style="list-style-type: none"> • The info "Resolved by library profile" means that a (Beckhoff) placeholder is resolved according to the specification in the library and thus according to the default resolution. • The info "Resolved by default library" means that a placeholder is resolved according to its default resolution ("Always newest"/""). • The info "Resolved by placeholder redirection" means that the resolution of a placeholder has been changed in this project. The default resolution is not used, but rather the resolution set by the user himself and saved in the PLC project file.

13.5.2 Changing the placeholder resolution

There are two ways to set the library to which a placeholder should be resolved:

- via the **Placeholder** dialog
- via the **Properties** window

You can use the placeholder dialog to set all placeholders in the project, that is, both the placeholders that are directly integrated at application level and the placeholders that are used within referenced libraries.

The properties window can only be used to set the placeholders available at application level.

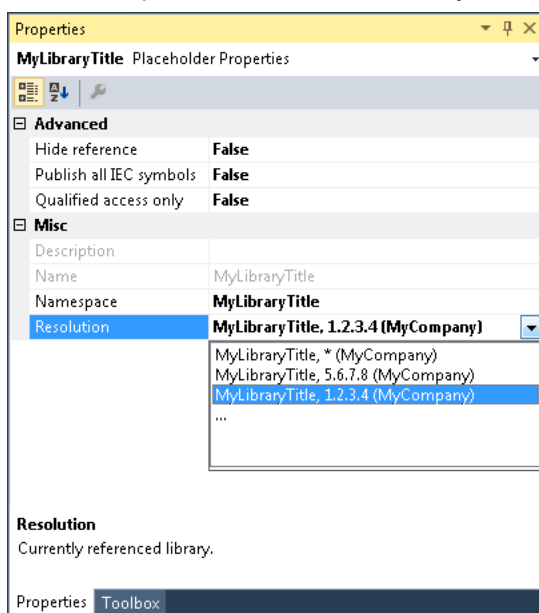
Possibility 1: Via the Placeholder dialog

- Open the [Placeholder \[▶ 280\]](#) dialog.
- Double-click the field in the **Library** column that belongs to the row of the placeholder whose resolution you want to change.
- A dialog through which you can select the desired library opens below the selected row. The dialog offers the following options.

Reset to default	This command is only available for placeholders that cannot currently be resolved according to their default resolution. This is the case, for example, if you add a Beckhoff placeholder and change the placeholder resolution to another library version. Using the command Reset to default you can resolve such a placeholder in accordance with the default resolution again. In doing so the library of the same name is usually selected with "Always newest"/"***".
Other versions of the selected library	If this area displays the library version to which you want to resolve the placeholder, you can click this version. The placeholder is then resolved to the same library, but to a different version of this library.
Other library	If you want to resolve a placeholder to another library, you can open a dialog via Other library , in which you can select the desired library from all installed libraries. If you activate the Display all versions option in this dialog, you can not only specify the desired library, but also select a specific version of this library.

Possibility 2: via the Properties window

- Open the [Properties \[▶ 362\]](#) window.
- Select a placeholder in the project tree below the References node.
- The Properties window shows the properties of this placeholder.
- The **Resolution** field provides a drop-down list containing the available versions of the previously selected library and the entry "...".
 - If this area displays the library version to which you want to resolve the placeholder, you can click this version. The placeholder is then resolved to the same library, but to a different version of this library.
 - If you want to resolve the placeholder to another library, you can open a dialog via "...", in which you can select the desired library from all installed libraries. If you activate the **Display all versions** option in this dialog, you can not only specify the desired library, but also select a specific version of this library.



13.6 Library documentation

You can use comments to document library objects and individual elements of library objects to explain the function and use of the program elements provided by the library.

The documentation created via the comments is displayed as text or as a table in the Library Manager, in which the library is integrated, on the **Inputs/Outputs** and **Library Parameters** tabs (description of individual elements of library objects) and in the **Documentation** tab (general description of library objects). (see [Basics \[▶ 283\]](#))

● TE1030 | TwinCAT 3 Documentation Generation

i For a more attractive presentation and a larger range of functions in the area of documentation, we recommend using the TE1030 | TwinCAT 3 Documentation Generation tool.

● Support for reStructuredText up to and including TwinCAT 3.1 Build 4024

i Up to and including Build 4024, the documentation format reStructuredText is available as an alternative.

See also:

- [Library Manager \[► 276\]](#)

13.6.1 Basics

TwinCAT offers the option of documenting library objects with the aid of comments. You can use comment for the library objects themselves and for the individual elements of the library objects:

- [General description of library objects \[► 283\]](#)
- [Description of individual elements of library objects \[► 284\]](#)

You can use the comments to describe the purpose of the object/element and how it can or should be used by the library user. The documentation is displayed in the Library Manager in which the library is integrated. The comments can be single-line (comment operator "//...") or multi-line (comment operator "(*...*)").

General description of library objects

You can describe library objects by adding a comment above the declaration line of the library object. If there are attributes above the declaration line, you can position the comment either above or below the attributes.

The general description of library objects is displayed in the **Documentation** tab of the [Library Manager \[► 276\]](#).

The following objects can be described using a general comment:

- POU objects (program, function block, function)
- Interfaces
- Methods, properties
- Global variable lists, parameter lists
- DUT objects (enumeration, structure, union, alias)

Samples for positioning the comments:

General description of a function block

```
// General FB comment
FUNCTION_BLOCK FB_Lib
VAR
...

```

General description of a global variable list

```
// General GVL comment. The comment can be placed over possible attributes.
{attribute 'qualified_only'}
VAR_GLOBAL
    fGlobal1      : REAL;
END_VAR

```

General description of a parameter list

```
{attribute 'qualified_only'}
// General parameter list comment. The comment can be placed below possible attributes.
VAR_GLOBAL CONSTANT
    cParameter1  : LREAL := 12.34;
END_VAR

```


Extended library documentation (reStructuredText)

If you select the documentation format **reStructuredText** in the project properties of the library and mark the comment text for the general description of library objects according to the syntax of reStructuredText, you have many more documentation options. (See [Extended – reStructuredText](#) [▶ 288])

Description of individual elements of library objects

In addition to the library objects themselves, you can describe the individual elements of the library object separately by adding a comment to each element. You can position the comments of individual variables either in the line above the variable declaration or in the same line after the variable declaration. You can comment on the return type of a method or function by inserting the comment in the method declaration line after the return type.

The table that is shown in the **Input/Output** and **Documentation** tabs of the [Library Manager](#) [▶ 276] clearly displays the documentation for the individual elements of a library object. If the library object is a parameter list, the table is displayed in the **Library Parameters** and **Documentation** tabs.

The table shows the comments for the following elements in the Library Manager:

- Comments for individual variables
 - Inputs/outputs of a POU object (program, function block, function)
 - Inputs/outputs of a method
 - Global variables (global variable list)
 - Global constants (global variable list, parameter list)
 - Variables of a DUT object (enumeration, structure, union variables)
- Comment of the return type (method, function)

Samples for positioning the comments:

Description of individual function block variables

```
FUNCTION_BLOCK FB_Lib
VAR_INPUT
    // Comment for the first input placed over the declaration
    bInput1 : BOOL;
    bInput2 : BOOL;           // Comment for the second input placed behind the declaration
END_VAR
```

Description of the return type of a method

```
// General method comment
METHOD SampleMethod : BOOL // Comment for the method's return value
VAR_INPUT
    bInput : BOOL;           // Comment for input variable
END_VAR
```

Samples

Structure

In the following sample, both the structure itself and the individual variables of the structure are assigned a comment. The comments are displayed in the Library Manager if the focus is on the structure in the lower part of the Library Manager.

```
// General structure comment
TYPE ST_Lib :
STRUCT
    bVar : BOOL := TRUE; // Comment for BOOL variable
    nVar : INT := 123;   (* Comment for INT variable that may also be
                        expanded over several lines. *)
END_STRUCT
END_TYPE
```


Inputs/Outputs		Documentation			
STRUCT ST_Lib					
Name	Type	Inherited from	Address	Initial	Comment
bVar	BOOL			TRUE	Comment for BOOL variable
nVar	INT			123	Comment for INT variable that may also be expanded over several lines.

Inputs/Outputs		Documentation			
STRUCT ST_Lib					
General structure comment					
Name	Type	Inherited from	Address	Initial	Comment
bVar	BOOL			TRUE	Comment for BOOL variable
nVar	INT			123	Comment for INT variable that may also be expanded over several lines.

Global variable list (GVL)

In the following sample, both the GVL itself and the individual variables of the GVL are assigned a comment. The comments are displayed in the Library Manager if the focus is on the GVL in the lower part of the Library Manager.

In this sample, the comment for the GVL itself is above the attribute used. Alternatively, the comment can also be positioned below the attribute.

```
(* General GVL comment.
The comment can be placed over possible attributes.
And it can be expanded over several lines. *)
{attribute 'qualified_only'}
VAR_GLOBAL
    fGlobal    : REAL := 12.5; // Comment for the global variable
END_VAR
VAR_GLOBAL CONSTANT
    cGlobal    : INT := 3;     // Comment for the global constant
END_VAR
```

Inputs/Outputs		Documentation			
VAR_GLOBAL GVL_Lib					
Name	Type	Inherited from	Address	Initial	Comment
fGlobal	REAL			12.5	Comment for the global variable
cGlobal	INT			3	Comment for the global constant

Inputs/Outputs		Documentation			
VAR_GLOBAL GVL_Lib					
General GVL comment. The comment can be placed over possible attributes. And it can be expanded over several lines.					
Name	Type	Inherited from	Address	Initial	Comment
fGlobal	REAL			12.5	Comment for the global variable
cGlobal	INT			3	Comment for the global constant

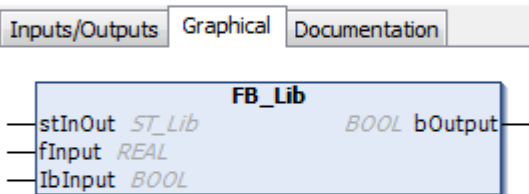
Function block with method

In the following sample, a function block has input/output variables and a method. The function block and the method itself, the individual input/output variables of the function block and of the method, and the return type of the method have a comment. The comments are displayed in the Library Manager if the focus is on the function block or the method in the lower part of the Library Manager.

Function block:

```
// General FB comment
FUNCTION_BLOCK FB_Lib
VAR_IN_OUT
  // Comment for the in-out-variable
  stInOut      : ST_Lib;
END_VAR
VAR_INPUT
  // Comment for this REAL input
  fInput       : REAL := 15.7;
  IbInput      AT%I* : BOOL;      // Comment for this allocated input
END_VAR
VAR_OUTPUT
  bOutput      : BOOL := TRUE;  // Comment for this output
END_VAR
VAR
END_VAR
```

Inputs/Outputs Graphical Documentation					
FUNCTION_BLOCK FB_Lib					
Name	Type	Inherited from	Address	Initial	Comment
stInOut	ST_Lib				Comment for the in-out-variable
fInput	REAL			15.7	Comment for this REAL input
IbInput	BOOL		%I*		Comment for this allocated input
bOutput	BOOL			TRUE	Comment for this output



Inputs/Outputs Graphical Documentation					
FUNCTION_BLOCK FB_Lib					
General FB comment					
Name	Type	Inherited from	Address	Initial	Comment
stInOut	ST_Lib				Comment for the in-out-variable
fInput	REAL			15.7	Comment for this REAL input
IbInput	BOOL		%I*		Comment for this allocated input
bOutput	BOOL			TRUE	Comment for this output

Method:

```
// General method comment
METHOD SampleMethod : BOOL // Comment for the method's return value
VAR_INPUT
  bInput       : BOOL;      // Comment for this BOOL input variable
  fInput       : LREAL;     // Comment for this LREAL input variable
END_VAR
```

Inputs/Outputs
Graphical
Documentation

METHOD SampleMethod

Name	Type	Inherited from	Address	Initial	Comment
SampleMethod	BOOL				Comment for the method's return value
bInput	BOOL				Comment for this BOOL input variable
fInput	LREAL				Comment for this LREAL input variable

Inputs/Outputs
Graphical
Documentation

SampleMethod

Inputs/Outputs
Graphical
Documentation

METHOD SampleMethod

General method comment

Name	Type	Inherited from	Address	Initial	Comment
SampleMethod	BOOL				Comment for the method's return value
bInput	BOOL				Comment for this BOOL input variable
fInput	LREAL				Comment for this LREAL input variable

Function block as subclass

In the following sample, a function block is declared as a subclass of the function block described above. The subclass itself and the additional input variable have a comment. These comments are displayed in the Library Manager if the focus is on the subclass in the lower part of the Library Manager. The variables of the superclass and their comments are also displayed for the subclass.

```

// General FB comment of the subclass
FUNCTION_BLOCK FB_Lib_Sub EXTENDS FB_Lib
VAR_INPUT
    bInputSub : BOOL;           // Comment for the input of subclass
END_VAR
VAR_OUTPUT
END_VAR
VAR
END_VAR
END_VAR
    
```

Inputs/Outputs
Graphical
Documentation

FUNCTION_BLOCK FB_Lib_Sub EXTENDS FB_Lib

Name	Type	Inherited from	Address	Initial	Comment
stInOut	ST_Lib	FB_Lib			Comment for the in-out-variable
fInput	REAL	FB_Lib		15.7	Comment for this REAL input
IbInput	BOOL	FB_Lib	%I*		Comment for this allocated input
bOutput	BOOL	FB_Lib		TRUE	Comment for this output
bInputSub	BOOL				Comment for the input of subclass

Inputs/Outputs
Graphical
Documentation

FB_Lib_Sub

Inputs/Outputs
Graphical
Documentation

FUNCTION_BLOCK FB_Lib_Sub EXTENDS FB_Lib

General FB comment of the subclass

Name	Type	Inherited from	Address	Initial	Comment
stInOut	ST_Lib	FB_Lib			Comment for the in-out-variable
fInput	REAL	FB_Lib		15.7	Comment for this REAL input
IbInput	BOOL	FB_Lib	%I*		Comment for this allocated input
bOutput	BOOL	FB_Lib		TRUE	Comment for this output
bInputSub	BOOL				Comment for the input of subclass

13.6.2 Extended – reStructuredText

● TE1030 | TwinCAT 3 Documentation Generation

i For a more attractive presentation and a larger range of functions in the area of documentation, we recommend using the TE1030 | TwinCAT 3 Documentation Generation tool.

● Support for reStructuredText up to and including TwinCAT 3.1 Build 4024

i Up to and including Build 4024, the documentation format reStructuredText is available as an alternative.

For a more attractive display of library object documentation in the Library Manager, you can select the documentation format **reStructuredText** in the project properties of the library (PLC Project Properties [▶_906] > category Common [▶_906]).

reStructuredText is an easy-to-read markup language that uses simple constructs to identify the structure of a document and special text elements such as section headings, bulleted lists and highlighting. More explicit constructs are used to include graphics and notes or to assign a function to text elements (hyperlinks).

Text elements that you mark in the comment above the declaration line of a library object according to the syntax of reStructuredText are structured and formatted in the **Documentation** tab of the Library Manager during library creation and optionally assigned a function.

The comment above the declaration line of the following objects is interpreted:

- POU objects (program, function block, function)
- Interfaces
- Methods, properties
- Global variable lists, parameter lists
- DUT objects (enumeration, structure, union, alias)

See also:

- [Library creation \[▶ 269\]](#)
- [Library Manager \[▶ 276\]](#)

13.6.2.1 Overview

Comments in reStructuredText consist of different (nested) text body elements and can be divided into sections with headings.

Starting with an introductory sample, the following sections provide an overview of the syntax of the reStructuredText markup.

In the first part of the description the essential concepts and syntax of reStructuredText are introduced. In the second, more application-oriented part of the description, various commenting options using different syntax elements are explained.



The correct use of the syntax is a prerequisite for error-free display in the Library Manager.

The information presented is based on the contents of the Dokutils documentation (reStructuredText markup specification).

Sample	<ul style="list-style-type: none"> • Samples [▶ 291]
General notes (Syntax elements [▶ 297])	<ul style="list-style-type: none"> • Blank lines and indentation [▶ 298] • Simple and more complex markup [▶ 301] • Explicit markup blocks [▶ 301] • Directives [▶ 302] • Inline markup [▶ 303] • Escaping mechanism [▶ 305] • Reference names [▶ 305]
Comment structure (Comment structure [▶ 306])	<ul style="list-style-type: none"> • Sections [▶ 306] • Transitions [▶ 309]
Text blocks (Text blocks [▶ 310])	<ul style="list-style-type: none"> • Text block (paragraph) [▶ 310] • Indented text block (block quote) [▶ 311] • Line-oriented text block (line block) [▶ 312]
Lists (Lists [▶ 314])	<ul style="list-style-type: none"> • Unordered enumeration list [▶ 314] • Ordered (numbered) enumeration list [▶ 315] • Definition list [▶ 316] • Field list [▶ 318]
Tables (Tables [▶ 319])	<ul style="list-style-type: none"> • Simple table [▶ 319] • Grid table [▶ 321] • CSV table [▶ 323] • List table [▶ 324]
Hyperlinks (Hyperlinks [▶ 324])	<ul style="list-style-type: none"> • Internal hyperlinks [▶ 326] • External hyperlinks [▶ 328] <ul style="list-style-type: none"> ◦ Standalone hyperlinks [▶ 330] ◦ Embedded URIs and aliases [▶ 330] • Indirect hyperlinks [▶ 331] • Inline hyperlinks [▶ 333] • Anonymous hyperlinks [▶ 334] • Footnotes [▶ 335] • Citations [▶ 343]
Substitution	<ul style="list-style-type: none"> • Substitution [▶ 344]
Note elements (Note elements [▶ 347])	<ul style="list-style-type: none"> • Specific notes [▶ 347] • Generic notes [▶ 348]
Images	<ul style="list-style-type: none"> • Images [▶ 349]
Code block	<ul style="list-style-type: none"> • Codeblock [▶ 355]
Internal comments	<ul style="list-style-type: none"> • Internal comments [▶ 356]
Font style (Font style [▶ 357])	<ul style="list-style-type: none"> • Highlighted text (italic) [▶ 357] • Highlighted text (bold) [▶ 357] • Inline literals (monospaced font) [▶ 357]

13.6.2.2 Samples

13.6.2.2.1 TwinCAT 3 sample project

This https://infosys.beckhoff.com/content/1033/tc3_plc_intro/Resources/7644044171/.zip can be used to download a TwinCAT 3 sample project that contains the code samples from this documentation as independent function blocks.

Save the PLC project as a library and include this library in a new project to compare the presentation of the documentation in the Library Manager with the original code. (See [Library creation](#) [▶ 269])

The "LibPOUs" folder of the sample project is divided as follows:

TC3 sample project		
Folder	Name of the function block	Documentation article
A_Samples		
	FB_DocuSample_FunctionBlock	Documentation of a function block [▶ 297]
	FB_DocuSample_SyntaxReminder	reStructuredText syntax reminder [▶ 293]
B_DocuElements		
Code block	FB_Libdoc_CodeBlock	Codeblock [▶ 355]
Comment structure	FB_Libdoc_Sections	Sections [▶ 306]
	FB_Libdoc_Transitions	Transitions [▶ 309]
Font style	FB_Libdoc_FontStyle	Font style [▶ 357]
Hyperlinks	FB_Libdoc_AnonymousHyperlinks	Anonymous hyperlinks [▶ 334]
	FB_Libdoc_Citation	Citations [▶ 343]
	FB_Libdoc_ExternalHyperlinks	External hyperlinks [▶ 328] Standalone hyperlinks [▶ 330] Embedded URIs and aliases [▶ 330]
	FB_Libdoc_IndirectHyperlinks	Indirect hyperlinks [▶ 331]
	FB_Libdoc_InlineHyperlinks	Inline hyperlinks [▶ 333]
	FB_Libdoc_InternalHyperlinks	Internal hyperlinks [▶ 326]
	FB_Libdoc_LinkToAnotherObject	Link to another object [▶ 344]
Hyperlinks\Footnotes	FB_Libdoc_AutomaticallyNumberedFootnotes	Automatically numbered footnotes [▶ 337]
	FB_Libdoc_AutomaticallySymbolFootnotes	Automatic generation of footnote symbols [▶ 341]
	FB_Libdoc_ManuallyAndAutomaticallyNumberedFootnotes	Manually and automatically numbered footnotes [▶ 342]
	FB_Libdoc_ManuallyNumberedFootnotes	Manually numbered footnotes [▶ 335]
	FB_Libdoc_NamedAutomaticallyNumberedFootnotes	Named automatically numbered footnotes [▶ 339]
Images	FB_Libdoc_Images	Images [▶ 349]
Internal comments	FB_Libdoc_InternalComments	Internal comments [▶ 356]
Lists	FB_Libdoc_DefinitionList	Definition list [▶ 316]
	FB_Libdoc_FieldList	Field list [▶ 318]
	FB_Libdoc_OrderedNumberedEnumerationList	Ordered (numbered) enumeration list [▶ 315]
	FB_Libdoc_UnorderedEnumerationList	Unordered enumeration list [▶ 314]
Note elements	FB_Libdoc_GenericNotes	Generic notes [▶ 348]
	FB_Libdoc_SpecificNotes	Specific notes [▶ 347]
Substitution	FB_Libdoc_Substitution_Images	Substitution [▶ 344]
	FB_Libdoc_Substitution_ReplacementText	Substitution [▶ 344]
Tables	FB_Libdoc_CSVTable	CSV table [▶ 323]
	FB_Libdoc_GridTable	Grid table [▶ 321]
	FB_Libdoc_ListTable	List table [▶ 324]
	FB_Libdoc_SimpleTables	Simple table [▶ 319]

TC3 sample project		
Folder	Name of the function block	Documentation article
Text blocks	FB_Libdoc_BlockQuote	Indented text block (block quote) [▶ 311]
	FB_Libdoc_LineBlock	Line-oriented text block (line block) [▶ 312]
	FB_Libdoc_Paragraph	Text block (paragraph) [▶ 310]

Documents about this

TC3_reStrText_Sample.zip (Resources/zip/9007206028223627.zip)

13.6.2.2 reStructuredText syntax reminder

The following sample gives an overview of the use of the reStructuredText syntax in the comment of a library object and the corresponding representation in the Library Manager.

(In [sample project](#) [▶ 291]: A_Samples\FB_DocuSample_SyntaxReminder)

Code

```
(*
=====
The reStructuredText Cheat Sheet: Syntax Reminder
=====
:Info: See <https://infosys.beckhoff.de/.
:Author: Beckhoff <support@beckhoff.com>
:Date: 2017-12-14

.. NOTE:: This syntax reminder is based on the docutils documentation

Section Structure
=====

A reStructuredText comment is made up of body elements, and may be
structured into sections. Section titles are underlined or overlined and underlined.
Sections contain body elements and/or subsections.

Paragraphs are flush-left and separated by blank lines.

Body Elements
=====
Grid table:

+-----+-----+
|Block quotes consist of |Literal block, preceded by "':":: | | | | |
|indented body elements: | |
|   Block quotes are indented. |   If (a=TRUE) Then |
| | | | |   b=3 |
| | | | | | |
+-----+-----+
|| Line blocks preserve line breaks and indents. |
||   Useful for adornment-free lists; |
||   long lines can be wrapped |
||   with continuation lines. |
+-----+-----+

Simple table:

=====
List Type      Examples
=====
Bullet list    - This is a bullet list
               - Items begin with "-", "+", or "*"
Enumerated list 1. This is an enumerated list.
                2. Items use any variation of "1.", "A)", and "(i)"
                #. Item can also be auto-enumerated
Definition list Term is flush-left : optional classifier
                Definition is indented, no blank line between
                continue
Field list     :Field name: field body
=====
```

```

Inline Markup
=====
*emphasis*, **strong emphasis**, ``inline literal text``

| Standalone hyperlink: http://beckhoff.de
| Named reference: Beckhoff_
| Anonymous reference: `Anonymous reference`__
| Footnote reference: [1]_
| Citation reference: [CIT2002]_
| Substitution: |substitution|
| Inline internal target: `Inline internal target`_

Explicit Markup
=====

=====
Explicit Markup      Examples (visible in the source code)
=====
Footnote            .. [1] Manually numbered or [#] auto-numbered
                   (even [#label]) or [*] auto-symbol
Citation            .. [CIT2002] This is a citation.
Hyperlink Target   .. _Beckhoff: http://beckhoff.de
                   .. _indirect target: Beckhoff_
                   .. _internal target:
Anonymous Target   __ http://beckhoff.de
Directive ("::")   .. image::, .. code::
Substitution Def   .. |substitution| replace:: This is a substitution
Comment            .. This text will not be shown
=====

Directives
=====

=====
Directive Name      Description
=====
attention           Specific admonition ("caution", "danger",
                   "error", "hint", "important", "note", "tip", "warning")
admonition          Generic titled admonition
image               .. image:: C:\Tc3LibDocImages\SampleLib1\logo.gif
                   :width: 40
code                .. code::

                   if(a=true) then b=3;
list-table          Create a table from a uniform two-level bullet list
csv-table           Create a table from CSV data
=====
*)

FUNCTION_BLOCK FB_DocuSample_SyntaxReminder
VAR_INPUT
    nVarInA : INT;          //First input variable
    nVarInB : INT := 5;    //Second input variable
END_VAR
VAR_OUTPUT
    nVarOut : INT;         //Output variable
END_VAR

```

Representation in the Library Manager

Inputs/Outputs Graphical Documentation

FB_DocuSample_SyntaxReminder (FB)

FUNCTION_BLOCK FB_DocuSample_SyntaxReminder

The reStructuredText Cheat Sheet: Syntax Reminder

Info: See <<https://infosys.beckhoff.de/>.
Author: Beckhoff <support@beckhoff.com>
Date: 2017-12-14

Note

This syntax reminder is based on the docutils documentation

Section Structure

A reStructuredText comment is made up of body elements, and may be structured into sections. Section titles are underlined or overlined and underlined. Sections contain body elements and/or subsections.

Paragraphs are flush-left and separated by blank lines.

Body Elements

Grid table:

Block quotes consist of indented body elements: Block quotes are indented.	Literal block, preceded by "``": <code>If (a=TRUE) Then b=3</code>
Line blocks preserve line breaks and indents. Useful for adornment-free lists; long lines can be wrapped with continuation lines.	

Simple table:

List Type	Examples
Bullet list	<ul style="list-style-type: none"> ▪ This is a bullet list ▪ Items begin with "-", "+", or "*"
Enumerated list	<ol style="list-style-type: none"> 1. This is an enumerated list. 2. Items use any variation of "1.", "A)", and "(i)" 3. Item can also be auto-enumerated
Definition list	Term is flush-left : <i>optional classifier</i> Definition is indented, no blank line between continue
Field list	Field name: field body

Inline Markup

emphasis, **strong emphasis**, `inline literal text`

Standalone hyperlink: <http://beckhoff.de>

Named reference: [Beckhoff](#)

Anonymous reference: [Anonymous reference](#)

Footnote reference: [\[1\]](#)

Citation reference: [\[CIT2002\]](#)

Substitution: This is a substitution

Inline internal target: [`Inline internal target`](#)

Explicit Markup

Explicit Markup	Examples (visible in the source code)
Footnote	[1] Manually numbered or [#] auto-numbered (even [#label]) or [*] auto-symbol
Citation	[CIT2002] This is a citation.
Hyperlink Target	

13.6.2.2.3 Documentation of a function block

The following sample shows the documentation of a function block using the reStructuredText syntax in the function block comment and the corresponding representation in the Library Manager.

(In [sample project \[▶ 291\]](#): A_Samples\FB_DocuSample_FunctionBlock)

Code

```
(*
:Description: This function block represents <...> and can be used for <...> ...
:Instructions for use: How to use this FB ...

Version history:
+-----+-----+-----+-----+-----+
|Date      | Version | created under | Author | Remark |
+-----+-----+-----+-----+-----+
|2017-07-04 | 1.0.0.0 | V3.1.4022.0 | S.H. | Performance optimization |
+-----+-----+-----+-----+-----+
|2019-01-11 | 1.1.0.0 | V3.1.4022.27 | K.F. | Bug fix: Output calculation corrected |
+-----+-----+-----+-----+-----+
*)

FUNCTION_BLOCK FB_Libdoc_FontStyle
VAR_INPUT
    nVarInA : INT; //First input variable
    nVarInB : INT :=5 //Second input variable
END_VAR
VAR_OUTPUT
    nVarOut : INT; //Output variable
END_VAR
```

Representation in the Library Manager

Inputs/Outputs
Graphical
Documentation

FB_DocuSample_FunctionBlock (FB)

FUNCTION_BLOCK FB_DocuSample_FunctionBlock

Description: This function block represents <...> and can be used for <...> ...

Instructions for use:
How to use this FB ...

Version history:

Date	Version	created under	Author	Remark
2017-07-04	1.0.0.0	V3.1.4022.0	S.H.	Performance optimization
2019-01-11	1.1.0.0	V3.1.4022.27	K.F.	Bug fix: Output calculation corrected

InOut:

Scope	Name	Type	Initial	Comment
Input	nVarInA	INT		First input variable
	nVarInB	INT	5	Second input variable
Output	nVarOut	INT		Output variable

13.6.2.3 Syntax elements

Different syntax elements and constructs are used in reStructuredText to mark words, phrases and paragraphs:

- [Blank lines and indentation \[▶ 298\]](#)
- [Simple and complex markup \[▶ 301\]](#)

- [Explicit markup blocks \[▶ 301\]](#)
 - [Directives \[▶ 302\]](#)
- [Inline markup \[▶ 303\]](#)

To obtain the standard meaning of the characters used for the markup, there is an escaping mechanism in reStructuredText:

- [Escaping mechanism \[▶ 305\]](#)

The identifiers of hyperlinks (reference names) are subject to certain requirements:

- [Reference names \[▶ 305\]](#)

13.6.2.3.1 Blank lines and indentation

The separation and nesting of text body elements is done in reStructuredText using the following design elements:

- [Blank lines \[▶ 298\]](#)
- [Paragraph and line breaks \[▶ 298\]](#)
- [Indentation \[▶ 298\]](#)

Blank lines

- Blank lines are used in a reStructuredText to separate paragraphs and other text body elements (see [Samples \[▶ 298\]](#)).
- Several consecutive blank lines are equivalent to a single blank line. (Exception: in a [Codeblock \[▶ 355\]](#) all blank lines are retained).
- Blank lines can be omitted if a markup or indentation makes the separation of text body elements unambiguous.
- The first line in the comment is treated as if it were preceded by a blank line. The last line in the comment is treated as if it were followed by a blank line.

Paragraph and line breaks

- New paragraphs are indicated by blank lines (see [Samples \[▶ 298\]](#)).
- Within [paragraphs \[▶ 310\]](#) the text is displayed continuously and automatically wrapped when the window width of the Library Manager is reached. Simple line breaks in the comment do not cause line breaks in the display. To implement line breaks, [line-oriented text blocks \(line blocks\) \[▶ 312\]](#) are used.

Indentation

- Indentations are used to identify nested contents (see [Samples \[▶ 298\]](#)).
- Each line of text whose indentation is smaller than that of the current level ends the current level of indentation.
- Spaces or tabs can be used for indentation. Since all indentations are significant, the degree of indentation must be consistent.

Samples

Text block (paragraph)

If a paragraph or an other construct consists of more than one line of text, the lines must be left aligned.

```
This is a paragraph. The lines of
this paragraph are aligned at the left.
```

```
    This paragraph has problems. The
lines are not left-aligned.
```

This is a paragraph. The lines of this paragraph are aligned at the left.

This paragraph has problems. The lines are not left-aligned.

See also: [Text block \(paragraph\) \[► 310\]](#)

Indented text block (block quote)

The indentation is the only marker for block quotes (indented text block).

- A blank line between a text block (paragraph or block quote) and a subsequent indented text block (block quote) is optional. The indentation of the following text block causes a break at the end of the previous text block. If a blank line is inserted in the comment, the line break occurs and the blank line remains in the display.
- A blank line is automatically displayed between an indented text block (block quote) and a subsequent text block that is not indented (paragraph or block quote), regardless of whether a blank line is inserted in the comment or not. The current level of indentation is ended.
- A blank line must be inserted in the comment between two text blocks (block quotes) on the same indentation level to separate the text blocks from each other.

```
This is a top-level paragraph.
```

```
    This paragraph belongs to a first-level block quote.
```

```
        This is the second paragraph of the first-level block quote.
```

```
            This paragraph belongs to a second-level block quote.
```

```
Another top-level paragraph.
```

```
    This paragraph belongs to a second-level block quote.
```

```
        This paragraph belongs to a first-level block quote. The
            second-level block quote above is inside this first-level
            block quote.
```

This is a top-level paragraph.

This paragraph belongs to a first-level block quote.

This is the second paragraph of the first-level block quote.

This paragraph belongs to a second-level block quote.

Another top-level paragraph.

This paragraph belongs to a second-level block quote.

This paragraph belongs to a first-level block quote. The second-level block quote above is inside this first-level block quote.

See also: [Indented text block \(block quote\) \[► 311\]](#)

Line-oriented text block (line block)

A line break can be implemented with a line block. Individual lines can be indented by indenting.

```
| Each new line begins with
| a vertical bar ("|").
|   Line breaks and initial indents
|   are preserved.
```

```
| Continuation lines are wrapped
portions of long lines; they begin
with spaces in place of vertical bars.
```

Each new line begins with
a vertical bar ("|").

Line breaks and initial indents
are preserved.

Continuation lines are wrapped portions of long lines; they begin with spaces in place
of vertical bars.

See also: [Line-oriented text block \(line block\) \[▶ 312\]](#)

Simple and complex markup

Several constructs start with a markup. The body of the construct must then be indented relative to the markup.

For constructs with simple markup (unordered and ordered lists, footnotes, citations, hyperlink targets, directives and comments), the degree of indentation of the body is determined by the position of the first line of text starting on the same line as the markup.

```
- This is the first line of a bullet list
item's paragraph. All lines must align
relative to the first line. [1]_
```

```
    This indented paragraph is interpreted
    as a block quote.
```

Because it is not sufficiently indented,
this paragraph does not belong to the list
item.

```
.. [1] Here's a footnote. The second line is aligned
with the beginning of the footnote label. The ".."
marker is what determines the indentation.
```

- This is the first line of a bullet list item's paragraph. All lines must align relative to the first line. [1]

This indented paragraph is interpreted as a block quote.

Because it is not sufficiently indented, this paragraph does not belong to the list item.

[1] Here's a footnote. The second line is aligned with the beginning of the footnote label. The ".." marker is what determines the indentation.

For constructs with more complex markup, the indentation of the first line after the markup usually determines the left edge of the text body. If the markup is very long, it may be useful to start with the text body in the next line. The line after the markup must be indented by at least one space (minimum indentation).

```
:Field: This field has a short field name, so aligning the field
        body with the first line is feasible.
```

```
:This field has a long field name: It would
    be very difficult to align the field body with the left edge
    of the first line.
```

```
:This field also has a long field name:
    It would be very difficult to align the field body with the left edge
    of the first line. It may even be preferable not to begin the
    body on the same line as the marker.
```


Field: This field has a short field name, so aligning the field body with the first line is feasible.

This field has a long field name:
It would be very difficult to align the field body with the left edge of the first line.

This field also has a long field name:
It would be very difficult to align the field body with the left edge of the first line. It may even be preferable not to begin the body on the same line as the marker.

See also:

- [Lists \[▶ 314\]](#)
- [Explicit markup blocks \[▶ 301\]](#)

13.6.2.3.2 Simple and more complex markup

reStructuredText includes simple and more complex markups. In contrast to simple markups, more complex markups can contain any text.

Without markup:

Use	Markup	Sample
Text blocks (paragraphs) [▶ 310]	-	Paragraph

Simple markups are used for the following constructs:

Use	Markup	Sample
Indented text block (block quote) [▶ 311]	Simple indentation	Paragraph Block quote
Line-oriented text block (line block) [▶ 312]	Vertical bar	Line block
Unordered enumeration list [▶ 314]	Bullets	- Item
Ordered enumeration list [▶ 315]	Enumerator	1. Item
Simple table [▶ 319]	Horizontal bars	===== Column 1 ===== Line 1 -----
Grid table [▶ 321]	Horizontal and vertical bars	+-----+ Column 1 +-----+ Line 1 +-----+

More complex markups are used for the following constructs:

Use	Markup	Sample
Definition list [▶ 316]	Any text followed by a colon and indentation	Term 3 : classifier Definition 3
Field list [▶ 318]	Any text enclosed by colons	:Organization: Beckhoff Automation GmbH & Co. . KG

13.6.2.3.3 Explicit markup blocks

Explicit markup blocks are text blocks in reStructuredText,

- whose first line starts with two dots followed by a space ("explicit markup start")
- whose second and continuation lines (if present) are indented relative to the first one
- which end before a line that is not indented

Principle

```
+-----+-----+
|".. " |explicit markup |
+-----+  block      |
      |                |
      +-----+-----+
```

A blank line is required between explicit markup blocks and other text body elements (for example, a preceding paragraph with text). A blank line between explicit markup blocks is optional.

Sample:

```
Paragraph
.. [1] Body elements
.. [2] Body elements
```

Use

Explicit markup blocks are used for the following constructs:

Use	Sample
Internal hyperlink targets [▶ 327]	.. _target:
External hyperlink targets [▶ 329]	.. _target: http://www.beckhoff.de
Indirect hyperlink targets [▶ 332]	.. _target: hyperlink-reference_
Anonymous hyperlink targets [▶ 335]	.. __: Anonymous-hyperlink-target
Manually numbered footnotes [▶ 335]	.. [1] Body elements
Automatically numbered footnotes [▶ 337]	.. [#] Body elements
Automatic generation of footnote symbols [▶ 341]	.. [*] Body elements
Quote (target) [▶ 343]	.. [CIT] Citation
Substitution definition [▶ 345]	.. ref type: definition
Internal comments [▶ 356]	.. comment

Explicit markup blocks with additional properties:

- [Directives \[▶ 302\]](#)

13.6.2.3.4 Directives

Directives are explicit markup blocks. These are syntax elements that affect an entire paragraph.

Structure

- Directives consist of a directive mark followed by a directive block.
- The directive markup consists of an explicit markup start (".. "), the type of directive, two colons and a space.
- The directive block consists of three parts, with individual directives using any combination of these parts:
 - directive argument
 - directive options
 - directive content
- Directive arguments can be file system paths, title texts etc.

- The directive options are specified using field lists; the possible field names and contents depend on the type of directive.
- Directive arguments and options must form a coherent block starting with the first or second line of the explicit markup block.
- The start of the content block of the directive is indicated by a blank line.

Principle

```
+-----+-----+
|".. "|directive type"::" directive |
+-----+block |
      | |
      +-----+
+-----+-----+
|
```

A blank line is required between a directive and a preceding text body element (for example, a paragraph with text). A blank line between directives is optional.

Sample:

```
Paragraph
.. image:: C:\Tc3LibDocImages\SampleLib1\img11.jpg
.. image:: C:\Tc3LibDocImages\SampleLib1\img12.jpg
```

Use

The following text elements can be created using directives:

Use	Sample
Specific notes [▶ 347]	<code>.. note::</code>
Generic notes [▶ 348]	<code>.. admonition:: Title</code>
Images [▶ 349]	<code>.. image:: C:\Users\SampleUser\Documents\LibraryDocumentationpicture.png :height: 100 px :width: 200 px</code>
Codeblock [▶ 355]	<code>.. code:: Code block</code>
CSV table [▶ 323]	<code>.. csv-table:: Table :header: "Column 1", "Column 2" :widths: 50, 50 "Line 1.1","Line 1.2"</code>
List table [▶ 324]	<code>.. list-table:: Table :widths: 50 50 :header-rows: 1 * - Column 1 - Column 2 * - Line 1.1 - Line 1.2</code>

13.6.2.3.5 Inline markup

Inline markups are used in reStructuredText to mark words or phrases within a text block. Words and phrases can thus be formatted or provided with a function.

Inline markups consist of a start and end character, which enclose the word or phrase in question.

Use

Inline markups are used for the following constructs:

- Constructs with identical start and end characters:

Use	Sample
Highlighted text (italic) [► 357]	<i>*emphasized text*</i>
Highlighted text (bold) [► 357]	**strong text**
Inline literals (monospaced font) [► 357]	<code>`inline literals`</code>
Substitution references [► 345]	substitution reference

- Constructs with different start and end characters:

Use	Sample
Hyperlink reference (inline, internal, external, indirect) (Hyperlinks [► 324])	Word: Target_ Phrase: `Hyperlink target`_
Anonymous hyperlink reference [► 334]	Word: Target__ Phrase: `Hyperlink target`__
Embedded URIs and aliases [► 330]	Embedded URI: `Beckhoff home page <http://www.beckhoff.de>`_ Alias: `link <Beckhoff home page_>`_
Standalone hyperlinks [► 330]	http://www.beckhoff.de support@beckhoff.com
Footnote reference (Footnotes [► 335])	Manual numbering: [1]_ Automatic numbering: [#]_ Automatic symbol generation: [*]_ [CIT]_
Quote reference [► 343]	[CIT]_
Inline hyperlinks [► 333]	Word: _Inline-hyperlink-target Phrase: _`Inline target`

Rules for inline markup detection

Inline markups cannot be nested. Start and end characters of inline markups are only recognized if the following conditions are met:

1. No space may follow the start character
2. The end character must not be preceded by a space.
3. The start character must start a text block or must be directly preceded by a space or one of the characters - : / ' " < ([{ .
4. The end character must end a text block or it must be directly followed by a space or one of the characters " . , ; ! ? -)] } / \ > .
5. The end character must be separated from the start character by at least one character.
6. Neither the start nor the end character may be preceded by a backslash (except the end character of inline literals). A backslash preceding a start or end character disables markup detection, except for the end character of inline literals.
7. If a start character is directly preceded by one of the characters ' " ([{ < , the corresponding character ' ")] } > must not follow directly (not possible: "*"text"*", possible: (* (text) *)).

Inline markup at character level

It is possible to mark individual characters within a word with a backslash, so that any text can follow immediately after the inline markup.

```
Python ``list``\s use square bracket syntax.
```

Python `lists` use square bracket syntax.

The backslash disappears from the edited document. The word "list" appears in monospaced font, and the letter "s" follows immediately as normal text, without space.

Any text can be prepended to the inline markup by using backslashes and spaces.

```
Possible in *re*\ ``Structured`` *Text*, though not encouraged
```

Possible in `reStructuredText`, though not encouraged.

The backslashes and spaces between "re", "Structured" and "Text" disappear from the edited comment.



It is not recommended to use backslash for inline markings at the character level. Such use makes it difficult to read the unprocessed comment. Use this function sparingly and only where it is absolutely necessary.

See also: [Escaping mechanism \[▶ 305\]](#)

13.6.2.3.6 Escaping mechanism

The character set generally available for plain text documents is limited. For example, individual markup characters can already have a meaning in the written text and appear in the text without being intended as markup. To obtain the standard meaning of the characters used for the markup, there is an escaping mechanism in reStructuredText. A backslash ("\") is used as escape character.

The following applies:

- A backslash followed by any character (except for spaces in non-URI contexts) causes the character to represent the character itself and plays no role in the interpretation of markups. The backslash is removed from the output.
- A literal backslash is represented by two backslashes in a row.
- In non-URI contexts, a backslash followed by a space is removed from the comment. This allows inline marking at character level (see [Inline markup at character level \[▶ 305\]](#)).
- There are two contexts in which backslashes have no special meaning: code blocks and inline literals. In these contexts a single backslash represents a literal backslash without having to be duplicated (see [Codeblock \[▶ 355\]](#)).

13.6.2.3.7 Reference names

reStructuredText distinguishes between simple reference names and phrase references.

Simple reference names

Simple reference names are single words consisting of alphanumeric characters (letters or digits) and isolated (non-adjacent) hyphens, underscores, dots, colons and plus signs. Spaces or other characters are not allowed. Simple reference names are used for quotation labels and some hyperlinks:

```
Want to learn about MyFavoriteProgrammingLanguage_?
```

```
.. _MyFavoriteProgrammingLanguage: http://www.python.org
```

or

```
Want to learn about My_Favorite_Programming-Language_?
```

```
.. _My_Favorite_Programming-Language: http://www.python.org
```

Phrase references

"Phrase references" are reference names that use punctuation or whose names are phrases (two or more words separated by spaces). Phrase references are expressed by including the phrase in back quotes and treating the text as a reference name:

```
Want to learn about `my favorite programming language`_?
.. _my favorite programming language: http://www.python.org
```

Want to learn about [my favorite programming language?](http://www.python.org)

Properties

Reference names are space-neutral and font-independent. If the reference names are resolved internally, the following applies:

- Spaces are normalized (one or more spaces and line breaks are interpreted as a single space)
- Upper and lower case is normalized (all letters are converted to lower case)

For example, the following hyperlink references are equivalent:

```
- `A HYPERLINK`_
- `a   hyperlink`_
- `A
  Hyperlink`_
.. _A HYPERLINK: https://beckhoff.de/
```

By clicking on one of the hyperlink references, the Beckhoff website is called up in the Library Manager.

- [A HYPERLINK](#)
- [a hyperlink](#)
- [A Hyperlink](#)

Hyperlinks, footnotes and citations have the same namespace for reference names. The names of quotations (simple reference names) and manually numbered footnotes (decimal numbers) are entered in the same database for the respective library object as other hyperlink names.

This means that within a library object a footnote (defined as `.. [1]`), which is normally referred to by a footnote reference (`[1]_`), is also referred to by a simple hyperlink reference (`1_`). Make sure that there are no conflicts with reference names.

See also: [Ambiguity in implicit and explicit hyperlinks within a library object](#) [► 325]

13.6.2.4 Comment structure

Comments can be structured in different ways.

On the one hand, comments can be divided into sections, which can be further divided into sections and/or contain text body elements. Sections are identified by a title.

On the other hand, paragraphs and text body elements can be separated by transitions in a comment. Separators are used to divide the text.

13.6.2.4.1 Sections

Sections are identified by a title and numbering. The title text is marked with "underscores" or "underscores" and suitable "overlines" in the comment.

Properties

- An underscore/overline is a single character that starts in the first column and forms a line that extends at least to the right margin of the title text. If overlines are used, the length and characters must match the underscores.
- The format of title texts that only have underscores or underscores and overlines differs, even if the same characters are used.
- An underscore/overline character can be any non-alphanumeric 7-bit ASCII character. The following characters are valid underscore/overline characters for section titles:
!"#\$%&'()*+,-./:;<=>?@[\\]^_`{|}~
The following characters are recommended:
"=", "_", "*"
- The number of section title levels is limited to five.
- The underscore/overline character does not determine the format of the title. This results from the order in which the titles with underscore or underscore and overline are found. The first style encountered is the outermost title, the second style is a subtitle, the third a subtitle and so on. Titles that are underscored with simple hyphens ("-") are always displayed as first level titles in gray and bold, regardless of their position.

<pre> Section Title ----- 1. Section Title ***** 1.1 Section Title ===== ----- 1.2.1 Section Title ----- ===== 1.2.1.1 Section Title ===== ===== 1.2.1.2 Section Title ===== ----- 1.2.2 Section Title ----- 1.2 Section Title ===== 2. Section Title ***** 2.1 Section Title ===== </pre>	<pre> Section Title ----- ===== 1. Section Title ===== ----- 1.1 Section Title ----- ----- 1.2.1 Section Title ----- ----- 1.2.1.1 Section Title ***** ----- 1.2.1.2 Section Title ***** ----- 1.2.2 Section Title ===== ----- 1.2 Section Title ----- ===== 2. Section Title ===== ----- 2.1 Section Title ----- </pre>	<p>Both code executions lead to the following representation in the Library Manager:</p> <p>Section Title</p> <p>1. Section Title</p> <p>1.1 Section Title</p> <p>1.2.1 Section Title</p> <p>1.2.1.1 Section Title</p> <p>1.2.1.2 Section Title</p> <p>1.2.2 Section Title</p> <p>1.2 Section Title</p> <p>2. Section Title</p> <p>2.1 Section Title</p>
---	--	--

- Not all section title formats have to be used, nor does a specific style have to be used for section titles. However, a comment must be consistent in the use of section headings: Once a hierarchy of title styles has been created, sections must use this hierarchy.
- Each section title is automatically assigned a hyperlink target, which can be referenced. The reference name of the hyperlink corresponds to the text of the section heading (see [Implicit hyperlink targets](#) [▶ 325]).

Sample

The following sample shows the use of sections and headings in a reStructuredText comment. A blank line under a title is optional. All text blocks up to the next title of the same or higher level are included in a section (or subsection etc.).

(In [sample project](#) [▶ 291]: B_DocuElements\Comment structure\FB_Libdoc_Sections)

```
(*
Section Title
-----

This document consists of two main sections and several subsections. All text blocks up to the next
title of the same or higher level
are included in a section (or subsection, etc.).

=====
1. Section Title
=====

Sections are identified through their titles, which are marked up with adornment:
"underlines" below the title text, or underlines and matching "overlines" above the title.

-----
1.1 Section Title
-----

A document must be consistent in its use of section titles: once a hierarchy of title styles is esta-
blished,
sections must use that hierarchy.
Rather than imposing a fixed number and order of section title adornment styles,
the order enforced will be the order as encountered.
The first style encountered will be an outermost title, the second style will be a subtitle,
the third will be a subsubtitle, and so on.

1.2.1 Section Title
=====

Underline-only adornment styles are distinct
from overline-and-underline styles that use the same character.

1.2.1.1 Section Title
*****
An underline/overline is a single repeated punctuation character that begins in column 1
and forms a line extending at least as far as the right edge of the title text.

1.2.1.2 Section Title
*****

1.2.2 Section Title
=====

-----
1.2 Section Title
-----

=====
2. Section Title
=====

-----
2.1 Section Title
-----

*)
```


FB_Libdoc_Sections (FB)

FUNCTION_BLOCK FB_Libdoc_Sections

Section Title

This document consists of two main sections and several subsections. All text blocks up to the next title of the same or higher level are included in a section (or subsection, etc.).

1. Section Title

Sections are identified through their titles, which are marked up with adornment: “underlines” below the title text, or underlines and matching “overlines” above the title.

1.1 Section Title

A document must be consistent in its use of section titles: once a hierarchy of title styles is established, sections must use that hierarchy. Rather than imposing a fixed number and order of section title adornment styles, the order enforced will be the order as encountered. The first style encountered will be an outermost title, the second style will be a subtitle, the third will be a subsubtitle, and so on.

1.2.1 Section Title

Underline-only adornment styles are distinct from overline-and-underline styles that use the same character.

1.2.1.1 Section Title

An underline/overline is a single repeated punctuation character that begins in column 1 and forms a line extending at least as far as the right edge of the title text.

1.2.1.2 Section Title

1.2.2 Section Title

1.2 Section Title

2. Section Title

2.1 Section Title

InOut:

Scope	Name	Type
Input	nVarA	INT
	nVarB	INT
Output	nSum	INT

13.6.2.4.2 Transitions

Instead of subheadings, separators between paragraphs and text body elements can be used to mark text divisions or to indicate changes in subject or emphasis. The separation of paragraphs and text body elements using separators is called a transition.

Properties

- A transition should neither start nor end a comment or section. In addition, two transitions should not be directly below each other.
- The syntax for a transition marker is a horizontal line with four or more characters. Valid separators are all non-alphanumeric 7-bit ASCII characters (e.g. "-", "+", "*", "="). A blank line must be inserted before and after the transition marker.
- Unlike the section titles, no hierarchy of transition markers is created. Differences in the transition markers have no effect. It is recommended to use a uniform style.

Sample

(In [sample project \[▶ 291\]](#): B_DocuElements\Comment structure\FB_Libdoc_Transitions)

```
(*
A transition marker is a horizontal line
of 4 or more repeated punctuation characters.

-----

A transition should not begin or end a section or document,
nor should two transitions be immediately adjacent.
*)
```

A transition marker is a horizontal line of 4 or more repeated punctuation characters.

A transition should not begin or end a section or document, nor should two transitions be immediately adjacent.

13.6.2.5 Text blocks

The following types of text blocks are distinguished in a reStructuredText comment:

- [Text block \(paragraph\) \[▶ 310\]](#)
- [Indented text block \(block quote\) \[▶ 311\]](#)
- [Line-oriented text block \(line block\) \[▶ 312\]](#)

13.6.2.5.1 Text block (paragraph)

Simple text blocks with left aligned text and without explicit marking are called paragraphs in reStructuredText.

Properties

- Paragraphs are separated from each other and from other text body elements by blank lines.
- Paragraphs may contain inline markups.

Principle

```
+-----+
|paragraph      |
+-----+
+-----+
|paragraph      |
+-----+
```

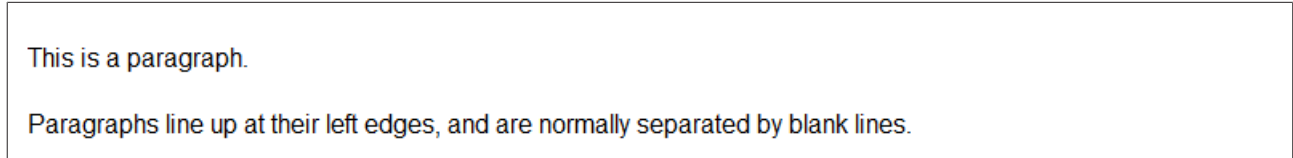
Sample

The following sample shows two paragraphs in a reStructuredText comment. Within paragraphs, the text is displayed continuously and automatically wrapped when the window width is reached.

(In [sample project \[▸ 291\]](#): B_DocuElements\Text blocks\FB_Libdoc_Paragraph)

```
(*
This is a paragraph.

Paragraphs line up at their left edges, and are normally separated
by blank lines.
*)
```



See also:

- [Blank lines and indentation \[▸ 298\]](#)
- [Indented text block \(block quote\) \[▸ 311\]](#)
- [Line-oriented text block \(line block\) \[▸ 312\]](#)

13.6.2.5.2 Indented text block (block quote)

In reStructuredText, text blocks that are indented relative to the preceding text without explicit marking are referred to as block quotes.

Properties

- The minimum indentation is one space.
- All markup processing (for text body elements and inline markups) continues within the block quote.

Unindented paragraph. Block quote 1. Block quote 2. Block quote 3.	Unindented paragraph. Block quote 1. Block quote 2. Block quote 3.
- List item. Block quote.	■ List item. Block quote.

Principle

```
+-----+
|(current level of) |
|indentation)      |
+-----+
      +-----+
      |block quote   |
      |(body elements) |
      +-----+
```

Sample

The following sample shows nested block quotes.

(In [sample project \[► 291\]](#): B_DocuElements\Text blocks\FB_Libdoc_BlockQuote)

- A blank line between a text block (paragraph or block quote) and a subsequent indented text block (block quote) is optional. The indentation of the following text block causes a break at the end of the previous text block. If a blank line is inserted in the comment, the line break occurs and the blank line remains in the display.
- A blank line is automatically displayed between an indented text block (block quote) and a subsequent text block that is not indented (paragraph or block quote), regardless of whether a blank line was inserted in the comment or not. The current level of indentation is ended.
- A blank line must be inserted in the comment between two text blocks (block quotes) on the same indentation level to separate the text blocks from each other.

```
(*
This is a top-level paragraph.

    This paragraph belongs to a first-level block quote.

    This is the second paragraph of the first-level block quote.

        This paragraph belongs to a second-level block quote.

Another top-level paragraph.

    This paragraph belongs to a second-level block quote.

    This paragraph belongs to a first-level block quote. The
    second-level block quote above is inside this first-level
    block quote.
*)
```

This is a top-level paragraph.

 This paragraph belongs to a first-level block quote.

 This is the second paragraph of the first-level block quote.

 This paragraph belongs to a second-level block quote.

Another top-level paragraph.

 This paragraph belongs to a second-level block quote.

 This paragraph belongs to a first-level block quote. The second-level block quote above is inside this first-level block quote.

See also:

- [Blank lines and indentation \[► 298\]](#)
- [Line-oriented text block \(line block\) \[► 312\]](#)

13.6.2.5.3 Line-oriented text block (line block)

Text blocks in which each line begins with the prefix "|" followed by a space are called line blocks in reStructuredText. Each vertical bar prefix indicates a new line.

Line blocks are suitable for structures where the line layout is important. In addition, a line block can be used to force blank lines in the Library Manager display or to separate text body elements.

Properties

- Indentations can result in nested structures.

- Wrapped sections of long lines are continuation lines. They start with a space instead of the vertical bar. The left margin of a continuation line must be indented, but does not have to be aligned with the left margin of the text above it.
- Inline markups are processed.
- A blank line must be inserted before and after a line block.

Principle

```
+----+-----+
| "| "|line      |
+----+-----+
      |continuation line |
      +-----+
```

Sample

The following sample shows the nesting of line blocks and the structure of continuation lines.
(In [sample project \[► 291\]](#): B_DocuElements\Text blocks\FB_Libdoc_LineBlock)

```
(*
| Each new line begins with
| a vertical bar ("|").
|     Line breaks and initial indents
|     are preserved.
| Continuation lines are wrapped
| portions of long lines; they begin
| with spaces in place of vertical bars.
*)
```

Each new line begins with
a vertical bar ("|").
Line breaks and initial indents
are preserved.
Continuation lines are wrapped portions of long lines; they begin with spaces in place of vertical bars.

If there is another text body element in front of a line block, the entire line block can be indented (the minimum indentation is one space).

```
(*
Line blocks are useful where the structure of lines is significant.

| Each new line begins with
| a vertical bar ("|").
|     Line breaks and initial indents
|     are preserved.
| Continuation lines are wrapped
| portions of long lines; they begin
| with spaces in place of vertical bars.

    | Each new line begins with
    | a vertical bar ("|").
    |     Line breaks and initial indents
    |     are preserved.
    | Continuation lines are wrapped
    | portions of long lines; they begin
    | with spaces in place of vertical bars.
*)
```

Line blocks are useful where the structure of lines is significant.

Each new line begins with a vertical bar ("|").

Line breaks and initial indents are preserved.

Continuation lines are wrapped portions of long lines; they begin with spaces in place of vertical bars.

Each new line begins with a vertical bar ("|").

Line breaks and initial indents are preserved.

Continuation lines are wrapped portions of long lines; they begin with spaces in place of vertical bars.

13.6.2.6 Lists

The following list types are distinguished in reStructuredText:

- [Unordered enumeration list \[► 314\]](#)
- [Ordered \(numbered\) enumeration list \[► 315\]](#)
- [Definition list \[► 316\]](#)
- [Field list \[► 318\]](#)

13.6.2.6.1 Unordered enumeration list

An unordered list consists of text blocks (bullets) beginning with a "-", "+" or "*", followed by a space.

Properties

- Regardless of the bullet type, a box is displayed as a bullet in the Library Manager.
- The asterisk "*" must not be used at the beginning of the comment.
- The list items are left aligned.
- The indentation of the bullet determines whether the entry is a main list or a sublist. The bullet character of a sublist begins at the level of the text of the list items in the main list.
- There is a space between the bullet and the text of the first line of a list item.
- If the text of a list item extends over several lines, the text of the continuation lines must begin at the level of the text of the first line.
- A blank line must be inserted before the first list item on a level. Blank lines between the individual list items and after the last list item on a level are optional.
- In the Library Manager, the main list and sublist are displayed as one block without blank lines.

Principle

```
+----+-----+
|"- "|list item      |
|   |(body elements) |
+----+-----|
```

Sample

The following sample shows an unordered enumeration list with two levels. There are two spaces before the list items in the sublist.

(In [sample project \[► 291\]](#): B_DocuElements\Lists\FB_Libdoc_UnorderedEnumerationList)

```
(*
This paragraph is not part of the list.

- This is the first bullet list item. The blank line above the
  first list item is required; blank lines between list items
  (such as below this paragraph) are optional.

- This is a sublist. The bullet lines up with the left edge
  of the text blocks above. A sublist is a new list so
  requires a blank line above. A blank line below is optional.

- This is a sublist. The bullet lines up with the left edge
  of the text blocks above. A sublist is a new list so
  requires a blank line above. A blank line below is optional.

- This is the second item of the main list.

- This is the third item of the main list.

This paragraph is not part of the list.
*)
```

This paragraph is not part of the list.

- This is the first bullet list item. The blank line above the first list item is required; blank lines between list items (such as below this paragraph) are optional.
 - This is a sublist. The bullet lines up with the left edge of the text blocks above. A sublist is a new list so requires a blank line above. A blank line below is optional.
 - This is a sublist. The bullet lines up with the left edge of the text blocks above. A sublist is a new list so requires a blank line above. A blank line below is optional.
- This is the second item of the main list.
- This is the third item of the main list.

This paragraph is not part of the list.

13.6.2.6.2 Ordered (numbered) enumeration list

An ordered (numbered) enumeration list consists of text blocks (bullets) starting with a number or a letter and a separator followed by a space. The list items follow a certain sequence.

Properties

- The list items are left aligned.
- The enumerator indentation determines whether the item is a main list or a sublist. The enumerator of a sublist starts at the level of the text of the list items in the main list.
- There is a space between the separator and the text.
- If the text of a list item extends over several lines, the text of the continuation lines must begin at the level of the text of the first line.
- A blank line must be inserted before the first list item on a level. Blank lines between the individual list items and after the last list item on a level are optional.
- In the Library Manager, the main list and sublist are displayed as one block without blank lines.
- The following enumerators are recognized:
 - Arabic numerals: 1, 2, 3, ... (no upper limit)
 - Capital letters: A, B, C, ..., Z
 - Lower-case letters: a, b, c, ..., z
- In addition, the auto-enumerator "#" can be used as a enumerator to automatically number a list. Automatically numbered lists can start with an explicit enumeration (Arabic numeral) that determines the sequence. Fully automatically numbered lists use Arabic numerals and start with 1.

- The following separators are recognized, but a dot is always displayed in the Library Manager:
 - appended with a dot: "1.", "A.", "a."
 - enclosed in brackets: "(1)", "(A)", "(a)"
 - followed by a right parenthesis: "1)", "A)", "a)"
- A new list is always started when
 - an enumerator is found that does not have the same format and sequence type as the current list (e.g. "1." "(a)" creates two separate lists);
 - the enumerations are not in the sequence (e.g. "1." and "3." create two separate lists).
- The second line of each enumeration is checked for validity. This is to prevent ordinary paragraphs from being misinterpreted as list items when they start with text that is identical to enumerators. This text is parsed as an ordinary paragraph, for example:

```
A. Einstein was a really
smart dude
```

- However, ambiguity cannot be avoided if the paragraph consists of only one line. This text is parsed as an enumeration in a list:

```
A. Einstein was a really smart dude
```

- If a single-line paragraph starts with a text identical to an enumerator ("A.", "1.", "(b)", etc.), the first character must be preceded by a backslash so that the line can be interpreted as a normal paragraph:

```
\A. Einstein was a really smart dude
```

Principle

```
+-----+-----+
|"1. " |list item   |
|      |(body elements) |
+-----+-----+|
```

Sample

The following sample shows an ordered (numbered) enumeration list with two levels. There are three spaces before the list items in the sublist.

(In [sample project \[► 291\]](#): B_DocuElements\Lists\FB_Libdoc_OrderedNumberedEnumerationList)

```
(*
1) This is the first item in the list.

   (a) This is the first subitem (1a).
   (b) This is the second subitem (1b).

2) This is the second item in the list.

   (a) This is the first subitem (2a).
   (b) This is the second subitem (2b).

#) This item is auto-enumerated.
*)
```

1. This is the first item in the list.
 - a. This is the first subitem (1a).
 - b. This is the second subitem (1b).
2. This is the second item in the list.
 - a. This is the first subitem (2a).
 - b. This is the second subitem (2b).
3. This item is auto-enumerated.

13.6.2.6.3 Definition list

A definition list describes a list of definitions. Each entry in a definition list contains a term, a definition and optionally classifiers.

Definition lists can be used in different ways:

- As a dictionary or glossary. The term is the word itself, a classifier can be used to identify the use of the term (noun, verb etc.), followed by the definition.
- For describing program variables. The term is the variable name. A classifier can be used to specify the type of variable (string, integer etc.), and the definition describes how the variables are used in the program.

Properties

- A term is a simple one-line word or phrase.
- A definition is a block that is indented relative to the term (the minimum indentation is one space) and can contain several paragraphs and other body elements.
- Optional classifiers can follow the term on the same line, each after a " : " (space, colon, space).

Principle

```
+-----+
|term[" : "classifier]*      |
+---+-----+
      |definition            |
      |(body elements)      |
+-----+
```

Sample

The sample shows a definition list with terms, classifiers and definitions.

(In [sample project \[► 291\]](#): B_DocuElements\Lists\FB_Libdoc_DefinitionList)

- Further text body elements are used within the definition list (paragraph, unordered list items).
- There must not be a blank line between a term line and a definition block (this distinguishes definition lists from block quotes).
- Blank lines are required before the first and after the last list item of the definition list, but can also be inserted between them.

```
(*
Term 1
  Definition 1.

Term 2
  Definition 2, paragraph 1.

  Definition 2, paragraph 2.

Term 3 : classifier
  Definition 3.

Term 4 : classifier one : classifier two
  - Item 1
  - Item 2
*)
```

Term 1
 Definition 1.

Term 2
 Definition 2, paragraph 1.

 Definition 2, paragraph 2.

Term 3 : *classifier*
 Definition 3.

Term 4 : *classifier one* : *classifier two*
 ■ Item 1
 ■ Item 2

13.6.2.6.4 Field list

Field lists are assignments of field names to field texts and can be used for two-column table-type structures.

Properties

- A field name can consist of any character and several words. Colons within a field name must be preceded by a backslash if they are followed by a space.
- Inline markups are processed in field names.
- The field name together with a colon prefix and suffix form the field marker. The field marker is followed by a space and the field body.
- The field body can contain several body elements that are indented relative to the field marker. The first line after the field marker determines the indentation of the field body (the minimum indentation is one space).

Principle

```
+-----+-----+
|": "field name": "| fieldbody |
+-----+-----+
| (body elements) |
+-----+-----+
```

Sample

(In [sample project \[▶ 291\]](#): B_DocuElements\Lists\FB_Libdoc_FieldList)

```
(*
:Organization: Beckhoff Automation GmbH & Co. KG
:Contact: info@beckhoff.de
:Address: | Hülshorstweg 20
          | 33415 Verl
          | Germany
:Authors: - Me
          - Myself
          - I
:Version: 1.0
>Status: released
>Date: 2017-12-07

:Copyright:
© Beckhoff Automation GmbH & Co. KG, Germany.
The reproduction, distribution and utilization of this document as well as
the communication of its contents to others without express authorization are prohibited.
Offenders will be held liable for the payment of damages.
All rights reserved in the event of the grant of a patent, utility model or design.
:Abstract: topic
:Indentation:
Since the field marker may be quite long, the second
and subsequent lines of the field body do not have to line up
with the first line, but they must be indented relative to the
field name marker, and they must line up with each other.
:LongLongFieldName: Since the field marker is quite long,
                    the field body is shown in the line after the marker.
:Parameter nIn: Integer
:Parameter nVarB: Integer
:*Parameter nVarC*: Integer
:``Parameter nVarD``: Integer
*)
```

Organization: Beckhoff Automation GmbH & Co. KG
Contact: info@beckhoff.de
Address: Hülshorstweg 20
 33415 Verl
 Germany

Authors:

- Me
- Myself
- I

Version: 1.0
Status: released
Date: 2017-12-07
Copyright: © Beckhoff Automation GmbH & Co. KG, Germany. The reproduction, distribution and utilization of this document as well as the communication of its contents to others without express authorization are prohibited. Offenders will be held liable for the payment of damages. All rights reserved in the event of the grant of a patent, utility model or design.

Abstract: topic
Indentation: Since the field marker may be quite long, the second and subsequent lines of the field body do not have to line up with the first line, but they must be indented relative to the field name marker, and they must line up with each other.

LongLongFieldname:
 Since the field marker is quite long, the field body is shown in the line after the marker.

Parameter nIn: Integer
Parameter nVarB: Integer
Parameter nVarC: Integer
Parameter nVarD: Integer

13.6.2.7 Tables

reStructuredText offers two syntaxes for delimiting table cells:

- [Simple table \[► 319\]](#)
- [Grid table \[► 321\]](#)

Directives offer further possibilities for table generation:

- [CSV table \[► 323\]](#)
- [List table \[► 324\]](#)

13.6.2.7.1 Simple table

Simple tables provide a compact and easy-to-understand, but limited row-oriented table display for simple data sets.

For a more complete table description, see [Grid tables \[► 321\]](#).

Properties

- Cell contents are typically individual paragraphs, although any text body elements can be displayed in most cells.
- Simple tables allow multi-line rows (in all but the first column) and the linking of columns, but not of rows.

- Simple tables are described with horizontal borders consisting of the characters "=" and "-". The equal sign ("=") is used for upper and lower table borders to separate optional headers from the table body. The hyphen ("-") is used to indicate column spacing in a single row by underlining the linked columns and can optionally be used for explicit and/or visual separation of rows.
- As with other text body elements, blank lines are required before and after tables. To separate two consecutive tables, a paragraph with text or a paragraph with a vertical bar must be inserted. The latter is removed from the output.
- Left table borders should be aligned with the left edge of preceding text blocks. If tables are indented, they are considered as part of a block quote. The minimum indentation is one space.

Samples

(In [sample project \[► 291\]](#): B_DocuElements\Tables\FB_Libdoc_SimpleTables)

A simple table starts with an upper border of equal signs and one or more spaces at each column boundary (two or more spaces are recommended). Regardless of the table width, the upper edge must cover all table columns completely.

There must be at least two columns in the table (to distinguish them from section headings). The upper border can be followed by headers. The last of the optional headers is underlined with "=", again with spaces at the column edges. There must not be a blank line below the separator for the header line. The lower boundary of the table consists of "=", including spaces at column boundaries.

For example, here is a truth table, a three-column table with a header row and four body rows:

A	B	A and B
False	False	False
True	False	False
False	True	False
True	True	True

Lines of hyphens "-" can be used to link adjacent columns. The lines must cover all columns and be oriented according to the defined column borders. Text lines with hyphens must not contain any other text. A hyphen applies to the row immediately above. For example, here is a table with connected columns in the header:

Inputs		Output
A	B	A or B
False	False	False
True	False	True
False	True	True
True	True	True

Each line of text must contain spaces at the column boundaries, unless the cells have been merged across the columns.

Each line of text starts a new table row, except if there is an empty cell in the first column. In this case, the text line is interpreted as a continuation line:

Inputs		Output
A	B	A or B
False	False	False
	False	True
False	True	True
True	True	True

To start a new row in a simple table without text in the first column, use

- an empty comment ("."), which is not displayed in the output, or
- a backslash ("\") followed by a space

Blank lines are allowed within simple tables. Their interpretation depends on the context. Blank lines between table rows are ignored. Blank lines within multi-line rows can separate paragraphs or other body elements within cells.

The right column is unlimited; the text can go beyond the table border (as indicated by the table borders). However, it is recommended that the frames are long enough to contain the entire text.

The following sample illustrates:

- Continuation lines (table line 2 consists of two text lines, table line 3 of three text lines and table line 4 of four text lines)
- Blank lines for separating paragraphs (line 3, column 2)
- Text that goes beyond the right margin of the table
- New rows that do not contain any text in the first column (row 4)

```
(*
=====
Col 1 Col 2
=====
1      Second column of row 1.
2      Second column of row 2.
       Second line of paragraph.
3      Second column of row 2.

       Second line of paragraph
4      - Second column of row 3.

       - Second item in bullet
       list (row 4, column 2).
\      Row 5; column 1 will be empty.
=====
*)
```

Col 1	Col 2
1	Second column of row 1.
2	Second column of row 2. Second line of paragraph.
3	Second column of row 3. Second line of paragraph
4	<ul style="list-style-type: none"> ■ Second column of row 4. ■ Second item in bullet list (row 4, column 2).
	Row 5; column 1 will be empty.

13.6.2.7.2 Grid table

Grid tables provide a complete table display. They allow any cell contents (text body elements) as well as the linking of rows and columns. However, it can be difficult to create grid tables, especially for simple data records.

A simpler (but limited) table representation can be found in section [Simple table \[▶ 319\]](#).

Properties

- Grid tables are described by a visual grid consisting of the characters "-", "=", "|" and "+". The hyphen ("-") is used for horizontal lines (line separators). The vertical bar ("|") is used for vertical lines (column separators). The plus sign ("+") is used for intersections of horizontal and vertical lines. The equal sign ("=") can be used to separate headers from the table body.

- As with other text body elements, blank lines are required before and after tables. To separate two consecutive tables, a paragraph with text or a paragraph with a vertical bar must be inserted. The latter is removed from the output.
- Left table borders should be aligned with the left edge of preceding text blocks. If tables are indented, they are considered as part of a block quote. The minimum indentation is one space.

Samples

The following sample illustrates:

- Merged columns and rows
- Different text body elements in cells

(In sample project [► 291]: B_DocuElements\Tables\FB_Libdoc_GridTable)

```
(*
+-----+-----+-----+-----+
| Header row, column 1 | Header 2 | Header 3 | Header 4 |
| (header rows optional) | | | |
+-----+-----+-----+-----+
| body row 1, column 1 | column 2 | column 3 | column 4 |
+-----+-----+-----+-----+
| body row 2 | Cells may span columns. | | |
+-----+-----+-----+-----+
| body row 3 | Cells may | - Table cells | |
+-----+-----+-----+-----+
| body row 4 | span rows. | - contain | |
| | | - body elements. | |
+-----+-----+-----+-----+
*)
```

Header row, column 1 (header rows optional)	Header 2	Header 3	Header 4
body row 1, column 1	column 2	column 3	column 4
body row 2	Cells may span columns.		
body row 3	Cells may span rows.	<ul style="list-style-type: none"> ■ Table cells ■ contain ■ body elements. 	
body row 4			

Sample of an unwanted interaction between grid character and cell text

The following table contains a cell in row 2, which ranges from column 2 to column 4:

```
+-----+-----+-----+-----+
| row 1, col 1 | column 2 | column 3 | column 4 |
+-----+-----+-----+-----+
| row 2 | | | |
+-----+-----+-----+-----+
| row 3 | | | |
+-----+-----+-----+-----+
```

When a vertical bar is used in the text of this cell, unintended effects may occur if it is inadvertently aligned to a column boundary:

```
+-----+-----+-----+-----+
| row 1, col 1 | column 2 | column 3 | column 4 |
+-----+-----+-----+-----+
| row 2 | Use the command ``ls | more``. | | |
+-----+-----+-----+-----+
| row 3 | | | |
+-----+-----+-----+-----+
```

Several solutions are possible to break the continuity of the grid structure.

One way is to move the text by adding an additional space before the cell text. The space is removed in the output.

```
+-----+-----+-----+-----+
| row 1, col 1 | column 2 | column 3 | column 4 |
+-----+-----+-----+-----+
| row 2 | Use the command ``ls | more``. | | |
+-----+-----+-----+-----+
```

```
+-----+-----+-----+
| row 3  |         |         |         |
+-----+-----+-----+
```

Another option is to add an additional row to row 2. The blank line is removed from the output.

```
+-----+-----+-----+
| row 1, col 1 | column 2 | column 3 | column 4 |
+-----+-----+-----+
| row 2        | Use the command ``ls | more``.  |
+-----+-----+-----+
| row 3        |         |         |         |
+-----+-----+-----+
```

13.6.2.7.3 CSV table

The `csv table` directive can be used to create a table from comma-separated values (CSV).

Description	The directive selection consists of an explicit markup start (".. ") followed by the type of directive (<code>csv table</code>) and two colons. (See also: Directives [▶ 302])
Principle	<code>.. csv-table::</code>
Properties	<ul style="list-style-type: none"> A blank line is required between the directive and a preceding text body element (for example, a paragraph with text).

Options

The row block can optionally contain a flat list of table options. The following options are recognized:

<code>widths : integer [integer...]</code>	weighting of column widths A list of relative column widths separated by commas or spaces. By default, the columns have the same width (100%/#columns).
<code>header rows : integer</code>	header table The number of rows to be used as table header. Default value is 0.
<code>stub columns : integer</code>	header column table The number of table columns to be used as table header. Default value is 0.
<code>header : CSV data</code>	Additional data for the table header that is added independently of and before each header line.

Sample

The following sample shows a header table with four columns and two rows.
(In [sample project \[▶ 291\]](#): `B_DocuElements\Tables\FB_Libdoc_CSVTable`)

```
(*
.. csv-table:: Property list
:header: "Items", "Property 1", "Property 2", "Property 3"
:widths: 10, 15, 15, 15

"Item 1", 1.67, angular, red
"Item 2", "not specified", round, blue
*)
```

or

```
(*
.. csv-table:: Property list
:header-rows: 1
:widths: 10, 15, 15, 15

"Items", "Property 1", "Property 2", "Property 3"
"Item 1", 1.67, angular, red
"Item 2", "not specified", round, blue
*)
```

Property list			
Items	Property 1	Property 2	Property 3
Item 1	1.67	angular	red
Item 2	not specified	round	blue

13.6.2.7.4 List table

The `list table` directive allows you to create a table of data in a two-level enumeration list.

Description	The directive selection consists of an explicit markup start (" <code>..</code> ") followed by the type of directive (<code>list-table</code>) and two colons. (See also: Directives [► 302])
Principle	<code>.. list-table::</code>
Properties	<ul style="list-style-type: none"> The sublists of the enumeration list must contain the same number of list items. A blank line is required between the directive and a preceding text body element (for example, a paragraph with text).

Options

Optionally, the list table row block can contain a flat list of table options. The following option is recognized:

<code>widths : integer [integer...]</code>	weighting of column widths A list of relative column widths separated by commas or spaces. By default, the columns have the same width (<code>100%/#columns</code>).
<code>header rows : integer</code>	header table The number of rows to be used as table header. Default value is 0.
<code>stub columns : integer</code>	header column table The number of columns to be used as table header. Default value is 0.

Sample

(In [sample project \[► 291\]](#): `B_DocuElements\Tables\FB_Libdoc_ListTable`)

```
(*
.. list-table:: Property list
   :widths: 50 50 50
   :header-rows: 1

   * - Items
     - Property 1
     - Property 2
   * - Item 1
     - angular
     - red
   * - Item 2
     - round
     - blue
*)
```

Property list		
Items	Property 1	Property 2
Item 1	angular	red
Item 2	round	blue

13.6.2.8 Hyperlinks

Hyperlinks refer to another location within or outside the comment. They usually consist of two parts: hyperlink reference (source) and hyperlink target. If the body of the text contains a source link, there must also be a target link somewhere else in the comment (exception: [detached hyperlinks \[► 330\]](#)).

reStructuredText distinguishes explicit, implicit and inline hyperlink targets.

It is also possible to set a link to the documentation of another library object that is also included in that library (see [Link to another object](#) [► 344]).

Explicit hyperlink targets

Explicit hyperlink targets refer to a section within the function block documentation or to an external page and can be linked together. They can be named or anonymized. Unlike the named hyperlinks, the reference name is not used with anonymous hyperlinks to match the reference with their target (see [Anonymous hyperlinks](#) [► 334]).

- [Internal hyperlinks](#) [► 326] (link to a position within the comment or within the function block documentation)
- [External hyperlinks](#) [► 328] (link to a website or email program)
 - [Embedded URIs and aliases](#) [► 330]
 - [Standalone hyperlinks](#) [► 330]
- [Indirect hyperlinks](#) [► 331] (linking of explicit hyperlink targets)

Inline hyperlink targets

Inline hyperlink targets refer to the current text of a comment or function block documentation.

- [Inline hyperlinks](#) [► 333]

Implicit hyperlink targets

Implicit hyperlink targets are generated by section headings, footnotes and citations. Unlike explicit hyperlink targets, section headings, footnotes and citations automatically generate a hyperlink target to themselves; they do not contain a link block in their definition. The reference name corresponds to the section heading or the footnote or quotation label. Otherwise, implicit hyperlinks behave identically to explicit hyperlinks.

- [Footnotes](#) [► 335]
- [Citations](#) [► 343]
- [Sections](#) [► 306]

Ambiguity in implicit and explicit hyperlinks within a library object

- Explicit and implicit hyperlink targets with the same reference name: The hyperlinks do not work.

Error message: Duplicate target name, cannot be used as a unique reference: "1" ("Beckhoff").

Sample:

```
This is an explicit internal hyperlink reference: 1_
For more information see [1]_
-----
.. _1:
This is an explicit internal hyperlink target.
.. [1] Footnote
```

or

```
See Beckhoff_.
This is an explicit hyperlink to Beckhoff_.
-----
.. _Beckhoff: http:\\www.beckhoff.de
```

```
Beckhoff
=====
```

- Duplicate implicit hyperlink targets: The hyperlinks do not work.

Error message: Duplicate target name, cannot be used as a unique reference: "chapter a".

Sample:

```
Chapter 1
=====
```

```
Chapter a
*****
```

```
Chapter 2
=====
```

```
Chapter a
*****
```

```
-----
```

```
See `Chapter a`_
```

- Duplicate explicit hyperlink targets: The hyperlinks do not work. Exception: Duplicated external hyperlink targets (identical reference names and referenced URIs) are conflict-free and are not removed.

Error message: Duplicate target name, cannot be used as a unique reference: "1".

Sample:

```
This in an explicit internal hyperlink reference: 1_
```

```
This in another explicit internal hyperlink reference: 1_
```

```
-----
```

```
.. _1:
```

```
This is an explicit internal hyperlink target.
```

```
.. _1:
```

```
This is another explicit internal hyperlink target.
```

See also: [Reference names \[► 305\]](#)

13.6.2.8.1 Internal hyperlinks

Internal hyperlinks make it possible to connect one position to another within a comment. The hyperlink target always points to the subsequent text body element.

Hyperlink reference

Description	<p>The hyperlink reference consists of a reference name followed by an underscore: <code>reference-name_</code></p> <p>Phrase references must be specified in back quotes: <code>`reference name`_</code></p> <p>(See also: Reference names [► 305])</p>
Start and end characters	<ul style="list-style-type: none"> • No start character, end character = "_" • Start character = "`", end character = "`_" (phrase references) <p>(See also: Inline markup [► 303])</p>

Hyperlink target

Description	<p>The hyperlink target consists of an explicit markup start (".. "), an underscore, the reference name and a colon:</p> <pre>.. _reference-name:</pre> <p>A phrase reference in the hyperlink target can optionally be included in back quotes:</p> <pre>.. _`reference name`:</pre> <pre>.. _reference name:</pre> <p>(See also: Explicit markup blocks [▶ 301], Reference names [▶ 305])</p>
Principle	<pre>+-----+-----+ ".. " "_name":" +-----+-----+ +-----+-----+</pre>
Properties	<ul style="list-style-type: none"> • Internal hyperlink targets have an empty link block. • A blank line is required between the explicit markup block and the subsequent text body element. Blank lines between explicit markup blocks are optional.

Sample

(In [sample project \[▶ 291\]](#): B_DocuElements\Hyperlinks\FB_Libdoc_InternalHyperlinks)

Simple internal hyperlink target

Clicking the hyperlink reference `target_` displays the text body element under `.. _target: .`

```
(*
Clicking on this internal hyperlink will take us to the target_
below.

.. _target:

The hyperlink targets above point to this paragraph.
*)
```

Clicking on this internal hyperlink will take us to the [target](#) below.

The hyperlink targets above point to this paragraph.

Nesting of internal hyperlink targets

The hyperlink also works if the internal hyperlink target is "nested" in an indented text block. For example, hyperlink targets can be set to individual list elements (with the exception of the first, since a previous internal hyperlink target applies to the entire list):

```
(*
Clicking on this internal hyperlink will take us to the `third item`_ of the bullet list.

* First list item
* Second list item

  .. _third item:

* Third list item, with hyperlink target.
*)
```

Clicking on this internal hyperlink will take us to the [third item](#) of the bullet list.

- First list item
- Second list item
- Third list item, with hyperlink target.

Concatenation of internal hyperlink targets

Internal hyperlink targets can be "concatenated". Several neighboring internal hyperlink targets then point to the same element:

```
(*
Clicking on this internal hyperlink will take us to target1_
and clicking on this internal hyperlink will take us to target2_.
Both targets point the the same paragraph.

.. _target1:
.. _target2:

The targets "target1" and "target2" are synonyms; they both
point to this paragraph.
*)
```

Clicking on this internal hyperlink will take us to [target1](#) and clicking on this internal hyperlink will take us to [target2](#). Both targets point to the same paragraph.

The targets "target1" and "target2" are synonyms; they both point to this paragraph.

Internal hyperlink targets can also be inserted in the current text (see: [Inline hyperlinks \[► 333\]](#)).

13.6.2.8.2 External hyperlinks

External hyperlinks make it possible to link from the comment or from the function block documentation to an external website or to open an email program.

Hyperlink reference

Description	<p>The hyperlink reference consists of a reference name followed by an underscore: reference-name_</p> <p>Phrase references must be specified in back quotes: `reference name`_</p> <p>(See also: Reference names [► 305])</p>
Start and end characters	<ul style="list-style-type: none"> • No start character, end character = "_" • Start character = "`", end character = "`_" (phrase references) <p>(See also: Inline markup [► 303])</p>

Hyperlink target

<p>Description</p>	<p>The hyperlink target consists of an explicit markup start (".. "), an underscore, the reference name, a colon, spaces and a link block:</p> <pre>.. _reference-name: link-block</pre> <p>A phrase reference in the hyperlink target can optionally be included in back quotes:</p> <pre>.. _`reference name`: link-block</pre> <pre>.. _reference name: link-block</pre> <p>(See also: Explicit markup blocks [▶ 301], Reference names [▶ 305])</p>
<p>Principle</p>	<pre>+-----+-----+ ".. " _"name": " link +-----+block +-----+-----+</pre>
<p>Properties</p>	<ul style="list-style-type: none"> External hyperlink targets have an absolute URI or an email address in their link block. The URI of an external hyperlink can either start in the same line as the explicit markup or in an indented text block immediately afterwards, without blank lines between them. If there are several lines in the link block, they are concatenated. Line breaks in the link block are removed. The following external reference targets are therefore equivalent: <pre>.. _one-liner: https://infosys.beckhoff.de/ .. _starts-on-this-line: http:// infosys.beckhoff.de/ .. _entirely-below: https://infosys. beckhoff.de/</pre> <p>If the URI of an external hyperlink target contains an underscore as the last character, this must be escaped so as not to be confused with an indirect reference target (see Escaping mechanism [▶ 305]).</p>

Sample

By clicking on the hyperlink reference, the website is called up directly in the Library Manager or a corresponding email program is opened.

(In [sample project \[▶ 291\]](#): B_DocuElements\Hyperlinks\FB_Libdoc_ExternalHyperlinks)

```
(*
See the Beckhoff_ home page for more information.

Please contact the `Beckhoff Support`_ for technical assistance.

.. _Beckhoff: http://www.beckhoff.de
.. _Beckhoff Support: support@beckhoff.com

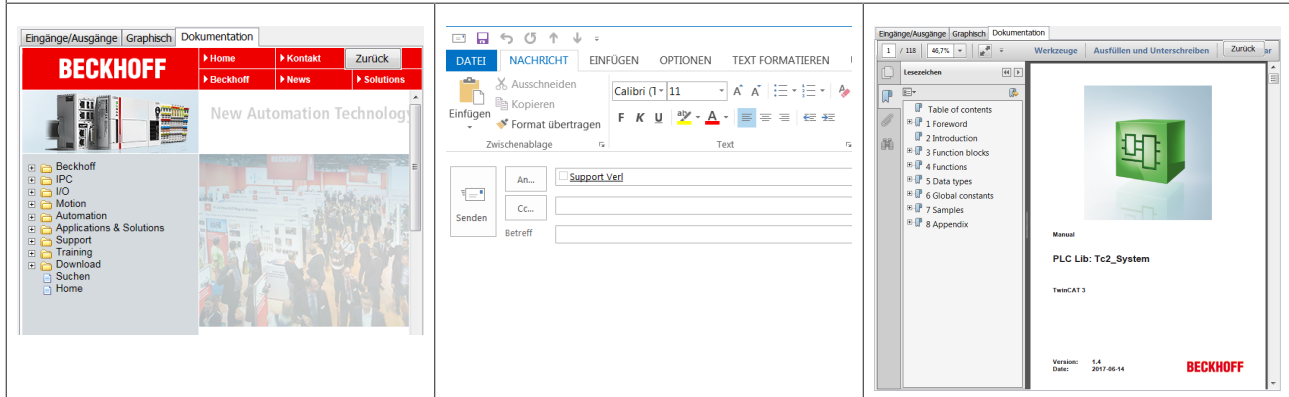
For more information see the `PLC Lib Tc2_System`_ and `PLC Lib Tc2_Standard`_.

.. _PLC Lib Tc2_System: https://download.beckhoff.com/download/document/automation/twincat3/
TwinCAT_3_PLC_Lib_Tc2_System_EN.pdf
.. _PLC Lib Tc2_Standard: https://download.beckhoff.com/download/document/automation/twincat3/
TwinCAT_3_PLC_Lib_Tc2_Standard_EN.pdf
*)
```

See the [Beckhoff](#) home page for more information.

Please contact the [Beckhoff Support](#) for technical assistance.

For more information see [PLC Lib Tc2_System](#) and [PLC Lib Tc2_Standard](#).



It is also possible to include URIs directly in references.

```
(*
External hyperlinks, like `Beckhoff <http://www.beckhoff.de/>`_.
*)
```

External hyperlinks, like [Beckhoff](#).

See also: [Embedded URIs and aliases \[► 330\]](#)

13.6.2.8.2.1 Standalone hyperlinks

A URI (Uniform Resource Identifier) within a text block is treated as a general external hyperlink, whereby the URI itself is displayed as link text.

Start and end characters	No start or end characters.
Properties	<ul style="list-style-type: none"> Absolute URIs and independent email addresses are recognized.

Sample

The following sample shows two single hyperlinks. The URI itself corresponds to the link text.
(In [sample project \[► 291\]](#): B_DocuElements\Hyperlinks\FB_Libdoc_ExternalHyperlinks)

```
(*
See http://www.beckhoff.de for info.
Contact support@beckhoff.com
*)
```

See <http://www.beckhoff.de> for info.

Contact support@beckhoff.com

13.6.2.8.2.2 Embedded URIs and aliases

Properties

- The parenthesized URI must be preceded by a space.
- The parenthesized URI must be the last text before the end of the string.

- With a single underscore at the end, the hyperlink reference is "named" and the same target URI can be referenced again.
- If two underscores are trailing, the hyperlink reference and hyperlink target are anonymous and the target cannot be referenced again. These are "unique" hyperlinks.

Example:

```
See the `Beckhoff Online Information System <http://infosys.beckhoff.de/`_ or the `Beckhoff home page <http://beckhoff.de/>`_.
```

This corresponds to:

```
See the `Beckhoff Online Information System`_ or the `Beckhoff home page`_.
_ http://infosys.beckhoff.de/
_ http://beckhoff.de/
```

See the [Beckhoff Online Information System](http://infosys.beckhoff.de/) or the [Beckhoff home page](http://beckhoff.de/).

The reference text can also be omitted, in which case the URI is duplicated for use as a reference text:

```
See the `http://infosys.beckhoff.de/`_ or the `http://beckhoff.de/`_.
```

See the <http://infosys.beckhoff.de/> or the <http://beckhoff.de/>.



The embedded URI construct facilitates creation and maintenance of hyperlinks at the expense of general readability. Inline URIs, especially long URIs, inevitably interrupt the natural text flow.

Sample

(In `sample project: B_DocuElements\Hyperlinks\FB_Libdoc_ExternalHyperlinks`)

A hyperlink reference can directly embed a target URI in angle brackets ("`<...>`").

```
See the `Beckhoff home page <http://www.beckhoff.de>`_ for info.
```

```
This `link <Beckhoff home page>`_ is an alias to the link above.
```

This corresponds to:

```
See the `Beckhoff home page`_ for info.
```

```
This link_ is an alias to the link above.
```

```
.. _Beckhoff home page: http://www.beckhoff.de
.. _link: `Beckhoff home page`_
```

See the [Beckhoff home page](http://www.beckhoff.de) for info.

This [link](#) is an alias to the link above.

13.6.2.8.3 Indirect hyperlinks

For indirect hyperlinks, the hyperlink target itself contains a hyperlink reference. Indirect hyperlinks thus enable the linking of explicit hyperlink targets.

Hyperlink reference

Description	<p>The hyperlink reference consists of a reference name followed by an underscore: reference-name_</p> <p>Phrase references must be specified in back quotes: `reference name`_</p> <p>(See also: Reference names [► 305])</p>
Start and end characters	<ul style="list-style-type: none"> • No start character, end character = "_" • Start character = "'", end character = "'_" (phrase references) <p>(See also: Inline markup [► 303])</p>

Hyperlink target

Description	<p>The hyperlink target consists of an explicit markup start (".. "), an underscore, the reference name, a colon, spaces and a link block:</p> <pre>.. _reference-name: link-block</pre> <p>A phrase reference in the hyperlink target can optionally be included in back quotes:</p> <pre>.. _`reference name`: link-block</pre> <pre>.. _reference name: link-block</pre> <p>(See also: Explicit markup blocks [► 301], Reference names [► 305])</p>
Principle	<pre>+-----+-----+ ".. " _"name":" link +-----+block +-----+-----+</pre>
Properties	<ul style="list-style-type: none"> • Indirect hyperlink targets have a hyperlink reference in their link block. • As with external hyperlink targets, the link block of an indirect hyperlink target can start in the same line as the explicit markup block or in the next line. <p>For example, the following indirect hyperlink targets are equivalent:</p> <pre>.. _one-liner: `A HYPERLINK`_ .. _entirely-below: `a hyperlink`_ .. _split: `A Hyperlink`_</pre> <p>If the reference name contains colons:</p> <ul style="list-style-type: none"> • the phrase must be enclosed in back quotes in the link block of the hyperlink target <pre>`Beckhoff Support`_ * worldwide support * design, programming and commissioning of complex automation systems * training program for Beckhoff system components .. _`Beckhoff Support`: support@beckhoff.com</pre> • or the colon(s) must be backslashed: <pre>`Beckhoff Support`_ * worldwide support * design, programming and commissioning of complex automation systems * training program for Beckhoff system components .. _Beckhoff Support\:: support@beckhoff.com</pre>

Samples

(In [sample project](#) [► 291]: B_DocuElements\Hyperlinks\FB_Libdoc_IndirectHyperlinks)

Indirect references to an internal reference target

In the following sample, the hyperlink target `.. _one` indirectly refers to target `.. _two` and the target `.. _two` indirectly refers to target `.. _three`, an internal reference target. Actually, all three objectives refer to the same target (the same paragraph):

```
(*
This hyperlink points to target one_ and indirect to target three.

.. _one: two_
.. _two: three_
.. _three:

The hyperlink targets above point to this paragraph.
*)
```

This hyperlink points to target [one](#) and indirect to target three.

The hyperlink targets above point to this paragraph.

Indirect references to an internal reference target

In the following sample, the target `.. _Beckhoff` indirectly refers to the target `.. _Beckhoff Information System`, an external reference target.

```
(*
The `Beckhoff Information System`_ is a reference source for Beckhoff_ products

.. _Beckhoff: `Beckhoff Information System`_
.. _Beckhoff Information System: https://infosys.beckhoff.de/
*)
```

The [Beckhoff Information System](#) is a reference source for [Beckhoff products](#)

It is also possible to insert an alias directly into the hyperlink target (see [Embedded URIs and aliases \[▶ 331\]](#)).

13.6.2.8.4 Inline hyperlinks

Inline hyperlinks correspond to internal hyperlinks, but the hyperlink target is inline in the text.

Hyperlink reference

Description	<p>The hyperlink reference consists of a reference name followed by an underscore: <code>reference-name_</code></p> <p>Phrase references must be specified in back quotes: <code>`reference name`_</code></p> <p>(See also: Reference names [▶ 305])</p>
Start and end characters	<ul style="list-style-type: none"> • No start character, end character = "_" • Start character = "'", end character = "'_" (phrase references) <p>(See also: Inline markup [▶ 303])</p>

Hyperlink target

Description	<p>The hyperlink target consists of an underscore followed by the reference name.</p> <pre>_reference-name</pre> <p>Phrase references must be specified in back quotes.</p> <pre>`_reference name`</pre> <p>Internal inline targets must not be anonymous.</p>
Start and end characters	<ul style="list-style-type: none"> • Start character = "_", no end character • Start character = "`_", end character = "`" <p>(See also: Inline markup [▶ 303])</p>

Sample

For example, the following paragraph contains a hyperlink reference and an inline hyperlink target called "FB_Sample". By clicking on the hyperlink reference, the sentence with the hyperlink target is displayed. (In [sample project \[▶ 291\]](#): B_DocuElements\Hyperlinks\FB_Libdoc_InlineHyperlinks)

```
(*
The function block `_FB_Sample` is used to...

For more information see the description of the `FB_Sample`_.
*)
```

The function block `FB_Sample` is used to...

For more information see the description of the [FB_Sample](#).

13.6.2.8.5 Anonymous hyperlinks

Designations of hyperlinks should generally be as detailed and meaningful as possible. However, duplicating a long reference name in the hyperlink target can be time-consuming and error-prone.

For anonymous hyperlinks, the hyperlink target does not contain a reference name, that is, the name of the hyperlink reference is not used to match the reference with its target. Instead, the order of anonymous hyperlink targets within the comment is significant: The first anonymous hyperlink reference is linked to the first anonymous hyperlink target, and so on. The number of anonymous hyperlink references in the comment must match the number of anonymous hyperlink targets.

Anonymous hyperlinks can lead to misleading and illegible comments. For reasons of readability, it is recommended that the hyperlink targets are placed as close as possible to the hyperlink references.



When editing comments with anonymous hyperlinks, note that the order of the corresponding hyperlink targets must also be adjusted, especially when adding, removing, and rearranging hyperlink references.

Hyperlink reference

Description	<p>The hyperlink reference consists of the reference name followed by two underscores:</p> <pre>anonymous-hyperlink-reference-name__.</pre> <p>Phrase references must be specified in back quotes.</p> <pre>`anonymous hyperlink reference name`__.</pre>
Start and end characters	<ul style="list-style-type: none"> • No start character, end character = "__" • Start character = "`", end character = "`__" (phrase references) <p>(See also: Inline markup [▶ 303])</p>

Hyperlink target

Description	<p>The hyperlink target consists of an explicit markup start (".. "), two underscores, a colon, spaces and a link block. There is no reference name.</p> <pre>.. __: anonymous-hyperlink-target-link-block</pre> <p>Alternatively, anonymous hyperlinks can consist of two underscores, a space and a link block:</p> <pre>__ anonymous-hyperlink-target-link-block</pre> <p>(See also: Explicit markup blocks [▶ 301])</p>
Principle	<pre>+-----+-----+ ".. " "__"name":" link +-----+block +-----+</pre>

Sample

In the following sample, the first anonymous hyperlink reference refers to the Beckhoff online information system, the second anonymous hyperlink reference refers to the Beckhoff website. If the order of the anonymous hyperlink targets is changed, the assignment of the references also changes.

(In [sample project \[▶ 291\]](#): B_DocuElements\Hyperlinks\FB_Libdoc_AnonymousHyperlinks)

```
(*
See the `Beckhoff Online Information System`__ or the `Beckhoff home page`__.
.. __: http://infosys.beckhoff.de/
.. __: http://beckhoff.de/
*)
```

See the [Beckhoff Online Information System](#) or the [Beckhoff home page](#).

13.6.2.8.6 Footnotes

Footnotes make it possible to remove annotations, remarks or references to a text passage from the text and thus make the text more legible.

Like hyperlinks, footnotes consist of two parts: Footnote reference (source) and footnote (target). Numeric names are used as reference names.

Types of footnotes

Footnotes can be numbered manually, automatically or mixed manually/automatically:

- [Manually numbered footnotes \[▶ 335\]](#)
- [Automatically numbered footnotes \[▶ 337\]](#)
- [Named automatically numbered footnotes \[▶ 339\]](#)
- [Manually and automatically numbered footnotes \[▶ 342\]](#)

It is also possible to automatically generate symbols for the footnotes:

- [Automatic generation of symbols \[▶ 341\]](#)

13.6.2.8.6.1 Manually numbered footnotes

For manually numbered footnotes, each footnote is assigned a name.

Footnote reference

Description	The footnote reference consists of a footnote label in square brackets followed by an underscore at the end. The footnote label is any integer number consisting of one or more digits.
Start and end characters	Start character = "[", end character = "]"_" (See also: Inline markup [► 303])

Footnote

Description	The footnote (footnote label) consists of an explicit markup start (".. "), the footnote label enclosed in square brackets, and a space followed by indented body elements (footnote content). (See also: Explicit markup blocks [► 301])
Principle	<pre>+-----+-----+ ".. " "["label"]" footnote +-----+-----+ (body elements) +-----+-----+</pre>
Properties	<ul style="list-style-type: none"> • There may or may not be a blank line between the footnote label and the footnote content. • The footnote content must be indented (at least one space). • Footnotes can be positioned anywhere in the comment, not just at the end. <p>Sample:</p> <pre>Footnote references, like [1]_. .. [1] Body elements go here.</pre> <p>or</p> <pre>Footnote references, like [1]_ .. [1] Body elements go here.</pre> <p>Footnote references, like [1].</p> <p>[1] Body elements go here.</p> <p>or</p> <pre>Footnote references, like [1]_. .. [1] - Body elements go here. - Second body element.</pre> <p>Footnote references, like [1].</p> <p>[1] ■ Body elements go here. ■ Second body element.</p>

Sample

Clicking the footnote reference [5]_ displays the footnote contents.

(In [sample project \[► 291\]](#):

B_DocuElements\Hyperlinks\Footnotes\FB_Libdoc_ManuallyNumberedFootnotes)

```
(*
**Numerical footnote**
```

```
Footnote references, like [5]_. Note that footnotes may get rearranged, e.g., to the bottom of the "
page".
```

```
-----
.. [5] A numerical footnote. Note there's no colon after the ``]``.
*)
```

Numerical footnote

Footnote references, like [5]. Note that footnotes may get rearranged, e.g., to the bottom of the "page".

[5] A numerical footnote. Note there's no colon after the] .

13.6.2.8.6.2 Automatically numbered footnotes

For automatically numbered footnotes, each footnote is automatically assigned a name.

Footnote reference

Description	The footnote reference consists of a footnote label in square brackets followed by an underscore at the end. The footnote label is a single "#".
Start and end characters	Start character = "[", end character = "]" (See also: Inline markup [▶_303])

Footnote

Description	<p>Each footnote (footnote label) consists of an explicit markup start (".. "), the footnote label enclosed in square brackets, and a space followed by indented body elements (footnote content).</p> <p>(See also: Explicit markup blocks [► 301])</p>
Principle	<pre>+-----+-----+ ".. " ["#"]" footnote +-----+ (body elements) +-----+ +</pre>
Properties	<ul style="list-style-type: none"> The first footnote to request automatic numbering is assigned the label "1", the second the label "2", etc. (if there are no manually numbered footnotes). A footnote that has automatically received a label "1" creates an implicit hyperlink target with the name "1", as if the label was explicitly specified. Numbering is determined by the order of the footnotes, not by the order of the footnote references. For automatically numbered footnotes ([#]_), the footnotes and footnote references must be in the same relative order but do not have to be linked. <p>Sample:</p> <pre>[#]_ is a reference to footnote 1, and [#]_ is a reference to footnote 2. .. [#] This is footnote 1. .. [#] This is footnote 2. .. [#] This is footnote 3. [#]_ is a reference to footnote 3.</pre> <p>[1] is a reference to footnote 1, and [2] is a reference to footnote 2.</p> <p>[1] This is footnote 1. [2] This is footnote 2. [3] This is footnote 3.</p> <p>[3] is a reference to footnote 3.</p> <p>Special caution is required if footnotes themselves contain auto-numbered footnote references or if several references are made in the immediate vicinity. Footnotes and references are noted in the order in which they appear in the comment, which does not necessarily correspond to the order in which a person would read them.</p> <p>(See also: Named automatically numbered footnotes [► 339])</p>

Sample

The following sample automatically shows numbered footnotes with and without an additional footnote label. (In [sample project \[► 291\]](#):

B_DocuElements\Hyperlinks\Footnotes\FB_Libdoc_AutomaticallyNumberedFootnotes)

```
(*
**Autonumbered footnotes**

Autonumbered footnotes are possible, like using [#]_ and [#]_.

-----

.. [#] This is the first one.

.. [#] This is the second one.
*)
```

Autonumbered footnotes

Autonumbered footnotes are possible, like using [1] and [2].

-
- [1] This is the first one.
[2] This is the second one.

13.6.2.8.6.3 Named automatically numbered footnotes

For named automatically numbered footnotes, each footnote is automatically assigned a name, and a label is explicitly specified at the same time.

Footnote reference

Description	The footnote reference consists of a footnote label in square brackets followed by an underscore at the end. The footnote label is a single "#" followed by any label.
Start and end characters	Start character = "[", end character = "]" (See also: Inline markup [▶_303])

Footnote

Description	Each footnote (footnote label) consists of an explicit markup start (".. "), the footnote label enclosed in square brackets, and a space followed by indented body elements (footnote content). (See also: Explicit markup blocks [► 301])
Principle	<pre>+-----+-----+ ".. " ["#label"]" footnote +-----+ (body elements) +-----+-----+</pre>
Properties	<ul style="list-style-type: none"> • A hyperlink target whose name corresponds to the specified label (without "#") is created on the footnote itself. • The explicit label allows an automatically numbered footnote to be uniquely assigned and designated multiple times as a footnote reference or hyperlink reference. <p>The sample shows an automatically numbered footnote that references both a footnote reference ([#note]_ or [#label]_) and a hyperlink reference (note_ or label_).</p> <p>If [#note]_ is the first footnote, it will show up as "[1]". We can refer to it again as [#note]_. We can also refer to it as note_ (an ordinary internal hyperlink reference).</p> <p>If [#label]_ is the second footnote, it will show up as "[2]". We can refer to it again as [#label]_. We can also refer to it as label_ (an ordinary internal hyperlink reference).</p> <pre>.. [#note] This is the first footnote labeled "note". .. [#label] This is the second footnote labeled "label".</pre> <p>If [1] is the first footnote, it will show up as "[1]". We can refer to it again as [1]. We can also refer to it as note (an ordinary internal hyperlink reference).</p> <p>If [2] is the second footnote, it will show up as "[2]". We can refer to it again as [2]. We can also refer to it as label (an ordinary internal hyperlink reference).</p> <pre>[1] (1, 2) This is the first footnote labeled "note". [2] (1, 2) This is the second footnote labeled "label".</pre> <p>(See also: Automatically numbered footnotes [► 337])</p>

Sample

The following sample automatically shows numbered footnotes with explicit footnote labels.

(In [sample project \[► 291\]](#):

B_DocuElements\Hyperlinks\Footnotes\FB_Libdoc_NamedAutomaticallyNumberedFootnotes)

```
(*
**Autonumbered labeled footnotes**

They may be assigned 'autonumber labels' - for instance, [#first]_ and [#second]_.

-----

.. [#first] a.k.a. first_
.. [#second] a.k.a. second_
*)
```


Autonumbered labeled footnotes

They may be assigned 'autonumber labels' - for instance, [1] and [2].

-
- [1] a.k.a. [first](#)
 - [2] a.k.a. [second](#)

13.6.2.8.6.4 Automatic generation of footnote symbols

When automatically generating footnote symbols, a symbol is assigned to each footnote.

Footnote reference

Description	The footnote reference consists of a footnote label in square brackets followed by an underscore at the end. The footnote label is a single asterisk "*".
Start and end characters	Start character = "[", end character = "]"_ " (See also: Inline markup [▶ 303])

Footnote

Description	Each footnote (footnote label) consists of an explicit markup start (".. "), the footnote label enclosed in square brackets, and a space followed by indented body elements (footnote content). (See also: Explicit markup blocks [▶ 301])
Principle	<pre> +-----+-----+ ".. " "["*""]" footnote +-----+ (body elements) +-----+ </pre>
Properties	<ul style="list-style-type: none"> • The number of footnote references must correspond to the number of footnotes. • A symbol footnote may not be referenced more than once. • A transformation inserts symbols as labels in the corresponding footnotes and footnote references. The following symbols are used for the footnotes: <ul style="list-style-type: none"> ◦ Asterisk/star ("*") ◦ Dagger ("†") ◦ Double dagger ("‡") ◦ Paragraph symbol ("§") ◦ Paragraph mark ("¶") ◦ Number sign ("#") ◦ Spades ("♠") ◦ Hearts ("♥") ◦ Diamonds ("♦") ◦ Clubs ("♣") • If more than ten symbols are needed, the same order is reused, doubled and then tripled, etc. ("***" etc.).

Sample

The following sample shows the automatic generation of footnote symbols and their representation in the Library Manager.

(In [sample project \[▶ 291\]](#):

B_DocuElements\Hyperlinks\Footnotes\FB_Libdoc_AutomaticallySymbolFootnotes)

```
(*
Here is a symbolic
footnote reference: [*]_ footnote reference: [*]_
footnote reference: [*]_ footnote reference: [*]_
footnote reference: [*]_ footnote reference: [*]_
footnote reference: [*]_ footnote reference: [*]_
footnote reference: [*]_ footnote reference: [*]_
footnote reference: [*]_
.. [*] This is the first footnote.
.. [*] This is the second footnote.
.. [*] This is the third footnote.
.. [*] This is the fourth footnote.
.. [*] This is the fifth footnote.
.. [*] This is the sixth footnote.
.. [*] This is the seventh footnote.
.. [*] This is the eighth footnote.
.. [*] This is the ninth footnote.
.. [*] This is the tenth footnote.
.. [*] This is the eleventh footnote.
*)
```

Here is a symbolic footnote reference: [❏] footnote reference: [⌈] footnote reference: [⌊] footnote reference: [§] footnote reference: [¶] footnote reference: [#] footnote reference: [♣] footnote reference: [♥] footnote reference: [♠] footnote reference: [♣] footnote reference: [❏]

[❏] This is the first footnote.
 [⌈] This is the second footnote.
 [⌊] This is the third footnote.
 [§] This is the fourth footnote.
 [¶] This is the fifth footnote.
 [#] This is the sixth footnote.
 [♣] This is the seventh footnote.
 [♥] This is the eighth footnote.
 [♠] This is the ninth footnote.
 [♣] This is the tenth footnote.
 [❏] This is the eleventh footnote.

13.6.2.8.6.5 Manually and automatically numbered footnotes

Both manually and automatically numbered footnotes can be used within a comment; the numbering of manually generated footnotes has priority. Only unused footnote labels are automatically assigned to numbered footnotes.

Sample

(In [sample project \[▶ 291\]](#): B_DocuElements

\Hyperlinks\Footnotes\FB_Libdoc_ManuallyAndAutomaticallyNumberedFootnotes)

```
(*
[2]_ will be "2" (manually numbered footnote),
[3]_ will be "3" (another manually numbered footnote),
[#]_ will be "4" (anonymous auto-numbered footnote), and
[#label]_ will be "1" (labeled auto-numbered).
-----
```

```
.. [2] This footnote is labeled manually, so its number is fixed.
.. [3] This footnote is also labeled manually, so its number is also fixed.
.. [#label] This autonumber-labeled footnote will be labeled "1".
It is the first auto-numbered footnote and no other footnote
with label "1" exists. The order of the footnotes is used to
determine numbering, not the order of the footnote references!
.. [#] This footnote will be labeled "4". It is the second
auto-numbered footnote, but footnote labels "1", "2", "3" are already used.
*)
```

[2] will be "2" (manually numbered footnote),
 [3] will be "3" (another manually numbered footnote),
 [4] will be "4" (anonymous auto-numbered footnote), and
 [1] will be "1" (labeled auto-numbered).

[2] This footnote is labeled manually, so its number is fixed.
 [3] This footnote is also labeled manually, so its number is also fixed.
 [1] This autonumber-labeled footnote will be labeled "1". It is the first auto-numbered footnote and no other footnote with label "1" exists. The order of the footnotes is used to determine numbering, not the order of the footnote references!
 [4] This footnote will be labeled "4". It is the second auto-numbered footnote, but footnote labels "1", "2", "3" are already used.

See also:

- [Manually numbered footnotes \[► 335\]](#)
- [Automatically numbered footnotes \[► 337\]](#)

13.6.2.8.7 Citations

Citations are identical to footnotes, except that they use alphanumeric names such as [note] or [GVR2001] and are displayed in tabular form in the Library Manager.

Like hyperlinks, citations consist of two parts: [citation reference \[► 343\]](#) (source) and [citation \[► 343\]](#) (target).

Quote reference

Description	Each citation reference consists of a citation label in square brackets followed by an underscore. Citation labels are simple reference names (single words, consisting of alphanumeric characters, no spaces).
Start and end characters	Start character = "[", end character = "]"_" (See also: Inline markup [► 303])

Quote (target)

Description	Each citation (citation label) consists of an explicit markup start (".. "), the citation label enclosed by square brackets and a space followed by indented body elements (citation content). (See also: Explicit markup blocks [► 301])
Principle	.. [CIT]
Properties	<ul style="list-style-type: none"> • Citations, unlike footnotes, are displayed in a table in the Library Manager. • There may or may not be a blank line between the citation label and the citation content. • The citation content must be indented (at least one space). • Citations can be positioned anywhere in the comment, not just at the end.

Sample

(In [sample project \[▶ 291\]](#): B_DocuElements\Hyperlinks\FB_Libdoc_Citation)

```
(*
Citation references, like [CIT2002]_. Note that citations may get rearranged, e.g., to the bottom of
the "page".

.. [CIT2002] This is the citation. It's just like a footnote,
   except the label is textual.

Given a citation like [this]_, one can also refer to it like this_.

.. [this] here.
*)
```

Citation references, like [\[CIT2002\]](#). Note that citations may get rearranged, e.g., to the bottom of the "page".

[\[CIT2002\]](#) This is the citation. It's just like a footnote, except the label is textual.

Given a citation like [\[this\]](#), one can also refer to it like [this](#).

[\[this\]](#) here.

13.6.2.8 Link to another object

It is possible to set a link to the documentation of another library object that is also included in that library. Clicking this link displays the documentation of the linked object or function block.

Description	The reference consists of :ref: followed by a reference name, which is specified in backquotes: :ref: `Objectname`
-------------	---

Sample

(In the [sample project \[▶ 291\]](#): B_DocuElements\Hyperlinks\FB_Libdoc_LinkToAnotherObject)

The following sample code contains two links, which refer to different function blocks. Clicking this link displays the documentation of the linked function block in the Library Manager.

```
(*
| It is also possible to create a reference to the documentation of another object which is also
| part of this library.
| For example, by clicking on this link :ref: `FB_Libdoc_InlineHyperlinks`, the documentation of this
| FB will be shown.
| Or you can also create a link to :ref: `FB_Libdoc_CodeBlock` which is part of another folder.
*)
```

13.6.2.9 Substitution

Substitutions allow you to include any number of complex inline structures in the text and at the same time keep the details out of the text flow.

They consist of two parts: [substitution reference \[▶ 345\]](#) and [substitution definition \[▶ 345\]](#). The processing system replaces the substitution references with the contents of the corresponding substitution definitions.

Substitution reference

Description	The substitution reference consists of a reference text enclosed by vertical bars. A substitution reference can also be a hyperlink reference by appending a "_" (named) or "__" (anonymous).
Start and end characters	Start character = " ", end character = " " (optionally followed by "_" or "__") (See also: Inline markup [▶ 303])
Properties	<ul style="list-style-type: none"> • Substitution references are replaced inline with the contents of the substitution definition. • A reference text must not start or end with a space

Substitution definition

Description	The substitution definition consists of an explicit markup start (".. ") followed by the reference text enclosed in vertical bars, a space and the definition block. The definition block contains an embedded inline-compatible directive, such as "image" or "replace". (See also: Explicit markup blocks [▶ 301])
Principle	<pre>+-----+-----+ ".. " " "reference text" " directive type"::" data +-----+directive block +-----+-----+</pre>
Properties	<ul style="list-style-type: none"> • The contents of the substitution definition replace the substitution reference inline. • A reference text must not start or end with a space.

Applications

Some use cases for the substitution mechanism are described below:

- [Replacement text \[▶ 345\]](#)
- [Images \[▶ 346\]](#)

Replacement text

The substitution mechanism can be used for simple text substitution. This can be useful if the replacement text is repeated several times in the comment, especially if it has to be changed later.

Directive type: `replace`

Samples

(In [sample project \[▶ 291\]](#): `B_DocuElements\Substitution\FB_Libdoc_Substitution_ReplacementText`)

```
(*
|RST|_ is an easy-to-read, what-you-see-is-what-you-get plaintext markup syntax.
Among others it is useful for library documentation. If the |RST| option is activated,
comments written in |RST|
will be shown in the Documentation tab of the library manager in an attractive format.
|RST| is a little annoying to type over and over and spelling out the
bicapitalized word |RST| every time isn't really necessary for |RST| source readability.

.. |RST| replace:: reStructuredText
.. _RST: https://infosys.beckhoff.de/
*)
```

`reStructuredText` is an easy-to-read, what-you-see-is-what-you-get plaintext markup syntax. Among others it is useful for library documentation. If the `reStructuredText` option is activated, comments written in `reStructuredText` will be shown in the Documentation tab of the library manager in an attractive format. `reStructuredText` is a little annoying to type over and over and spelling out the bicapitalized word `reStructuredText` every time isn't really necessary for `reStructuredText` source readability.

Note the final underscore when you first use the substitution reference. This indicates a reference to the corresponding hyperlink target. Inline markup is not processed in the substitution definition.

```
(*
This is a simple |substitution reference|. It will be replaced by
the processing system.

This is a |substitution and hyperlink reference|_. In
addition to being replaced, the replacement text or element will
refer to the "substitution and hyperlink reference" target.

.. |substitution reference| replace:: example of substitution
.. |substitution and hyperlink reference| replace:: combination of substitution and hyperlink
reference
.. _substitution and hyperlink reference: https://beckhoff.de/
*)
```

This is a simple example of substitution. It will be replaced by the processing system.

This is a [combination of substitution and hyperlink reference](#). In addition to being replaced, the replacement text or element will refer to the "substitution and hyperlink reference" target.

Images

(In [sample project \[▶ 291\]](#): B_DocuElements\Substitution\FB_Libdoc_Substitution_Images)

The substitution mechanism can also be used to include images in the comment text.

Directive type: `image`

(See also: [Images \[▶ 349\]](#))


Samples

Substitution in paragraphs

The comment text `|Beckhoff|_` is supplemented by the logo. The underscore following the substitution reference indicates a reference to a hyperlink target. Click on the image to open the Beckhoff website in the Library Manager.


```
(*
Images are a common use for substitution references: |Beckhoff|_.

.. |Beckhoff| image:: C:\Tc3LibDocImages\SampleLib1\logo.gif
:height: 16
:width: 64
.. _Beckhoff: http://www.beckhoff.de/
*)
```

Images are a common use for substitution references: 

```
(*
The |safety| symbol indicates a hazardous situation which,
if not avoided, could result in minor or moderate injury.
```

```
.. |safety| image:: C:\Tc3LibDocImages\logo\SampleLib1\safetysymbol.png
*)
```




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




Substitution in lists or tables

```
( *
* |Run mode| Run mode
* |Stop mode| Stop mode
* |Config mode| Config mode

=====
Symbol      Status
=====
|Run mode|  Run mode
|Stop mode| Stop mode
|Config mode| Config mode
=====

.. |Run mode| image:: C:\Tc3LibDocImages\SampleLib1\tc3rtmode.png
.. |Stop mode| image:: C:\Tc3LibDocImages\SampleLib1\tc3rtstopmode.png
.. |Config mode| image:: C:\Tc3LibDocImages\SampleLib1\tc3rtconfigmode.png
*)
```

-  Run mode
-  Stop mode
-  Config mode

Symbol	Status
	Run mode
	Stop mode
	Config mode

13.6.2.10 Note elements

reStructuredText distinguishes between specific and generic notes. Notes are displayed in the Library Manager as an offset block shaded with a title (signal word).

13.6.2.10.1 Specific notes

Specific notes (e.g. security and warning notices) can be generated with the directives `attention`, `caution`, `danger`, `error`, `hint`, `important`, `note`, `tip` and `warning`. The title of the note block is specified by the directive and corresponds to the type of note.

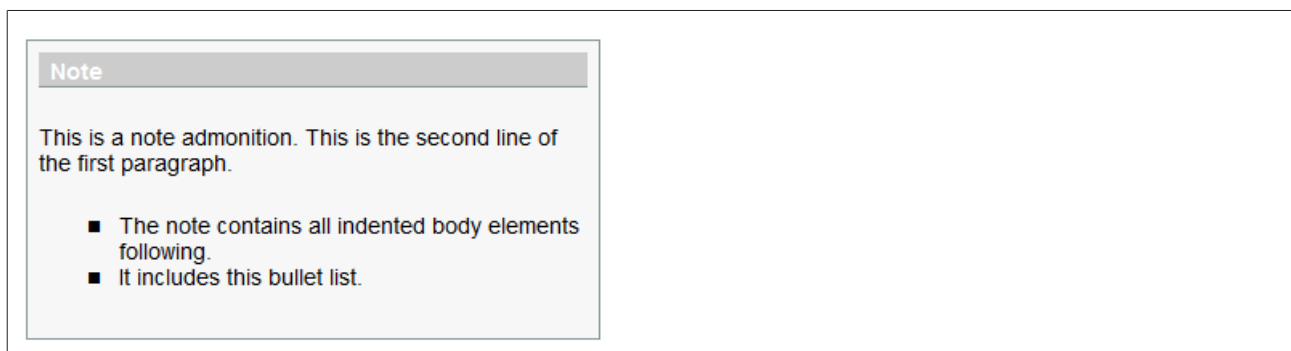
Description	The directive markup consists of an explicit markup start (".. ") followed by the type of directive (e.g. <code>note</code>) and two colons. (See also: Directives [▶ 302])
Principle	<code>.. note::</code>
Properties	<ul style="list-style-type: none"> • Specific notes can be used in the comment as ordinary body elements and contain any body elements. • The following note types have been implemented: <ul style="list-style-type: none"> ◦ Attention ◦ Caution ◦ Danger ◦ Error ◦ Hint ◦ Important ◦ Note ◦ Tip ◦ Warning • Any text immediately following the directive mark on the same line and/or indented on the following lines is interpreted as an instruction block (directive content). • A blank line is required between the directive and a preceding text body element (for example, a paragraph with text).

Sample

(In [sample project \[▶ 291\]](#): B_DocuElements\Note elements\FB_Libdoc_SpecificNotes)

```
(*
.. note:: This is a note admonition.
   This is the second line of the first paragraph.

- The note contains all indented body elements
  following.
- It includes this bullet list.
*)
```



13.6.2.10.2 Generic notes

Generic notes can be created with the `admonition` directive. For generic notes, you can choose the title and therefore the note type.

Description	The directive markup consists of an explicit markup start (".. ") followed by the type of directive (<code>admonition</code>) and two colons. (See also: Directives [▶ 302])
Principle	<code>.. admonition::</code>
Properties	<ul style="list-style-type: none"> • The directive argument corresponds to the reference title • A blank line must be inserted between the directive selection and the instruction block (directive content). • A blank line is required between the directive and a preceding text body element (for example, a paragraph with text).

Sample

(In [sample project \[▶ 291\]](#): B_DocuElements\Note elements\FB_Libdoc_GenericNotes)

```
(*
.. admonition:: And, by the way...

    You can make up your own admonition too.
*)
```



13.6.2.11 Images

Images can be included in the comment with the `image` directive.

Description	The directive markup consists of an explicit markup start (".. ") followed by the type of the directive (<code>image</code>) and two colons. (See also: Directives [▶ 302])
Principle	<code>.. image::</code>
Properties	<ul style="list-style-type: none"> • The file path for the image source file is specified in the argument of the directive. As with hyperlink targets, the file path can begin immediately after that in the same line as the explicit markup start or in an indented text block (without empty lines in between). The file path can be specified absolutely or relatively and may not contain any spaces. (See: Absolute or relative file path as directive argument [▶ 351]) • The image is displayed as wide as possible (=100%) according to the window width of the Library Manager. If the window width of the Library Manager is changed, the proportionality of the image is maintained. • A blank line is required between the directive and a preceding text body element (e.g. a paragraph with text).

Options

Optionally, the directive block can contain a flat field list with image options. The following options are recognized:

height: length	<p>Height of the image.</p> <p>Used to scale the image vertically. The width of the image is adjusted proportionally if possible.</p> <ul style="list-style-type: none"> • Specification in px (with or without spaces): <ul style="list-style-type: none"> ◦ The height of the displayed image always corresponds to the specified height of the image in px, regardless of the height and width of the Library Manager window. If the height of the Library Manager window is smaller than the specified image height, you can display the remaining part of the image by scrolling within the Library Manager. ◦ The image width is at most as large as the window width of the Library Manager: If the Library Manager window is wide enough, the image is displayed proportionally. Otherwise the display is distorted. ◦ The proportionality of the image therefore depends on the window width of the Library Manager.
width: length or percentage of the current line width	<p>Width of the image.</p> <p>Used to scale the image horizontally. The height of the image is adjusted proportionally if possible.</p> <ul style="list-style-type: none"> • Specification in px (with or without spaces): <ul style="list-style-type: none"> ◦ The width of the displayed image corresponds to the specified width of the image in px if possible but is at most as large as the window width of the Library Manager. This means that if the width of the Library Manager window is smaller than the specified image width, the image is only as wide as the Library Manager window. ◦ The height of the image is adjusted proportionally to the specified image width. ◦ The proportionality of the image therefore depends on the window width of the Library Manager. • Specification in % (with or without spaces): <ul style="list-style-type: none"> ◦ The width of the displayed image takes up the specified percentage of the window width of the Library Manager. ◦ The height of the image is adjusted proportionally to the image width. ◦ The proportionality of the image is thus maintained.
align: „left“ or „right“	<p>Alignment of the image.</p> <p>The values "left" and "right" control the horizontal alignment of an image so that the image can float and the text can flow around it.</p>
target: text (URI or reference name)	<p>Turns the image into a reference ("clickable"). The argument of the option can be a URI (relative or absolute) or a reference name with an underscore suffix (e.g. name_).</p>
scale: integer percentage	<p>Uniform scaling factor of the image.</p> <p>Used to scale the image proportionally.</p> <ul style="list-style-type: none"> • Specification in % (with or without spaces): <ul style="list-style-type: none"> ◦ The width and the height of the displayed image always correspond to the specified percentage of the original image size (i.e. the size of the graphic that is referenced in the file system), irrespective of the width and height of the Library Manager window. If the width or height of the Library Manager window is smaller than the specified image width or height, you can display the remaining part of the image by scrolling within the Library Manager. ◦ The proportionality of the image is thus maintained.

If an image is included without a height or width option, the width of the displayed image corresponds to the window width of the Library Manager. The height of the image is adjusted proportionally to the image width. The image is displayed as large as possible. The proportionality of the image is maintained.

Absolute or relative file path as directive argument

You can specify both absolute and relative file paths in the argument of the image directive in order to reference images.

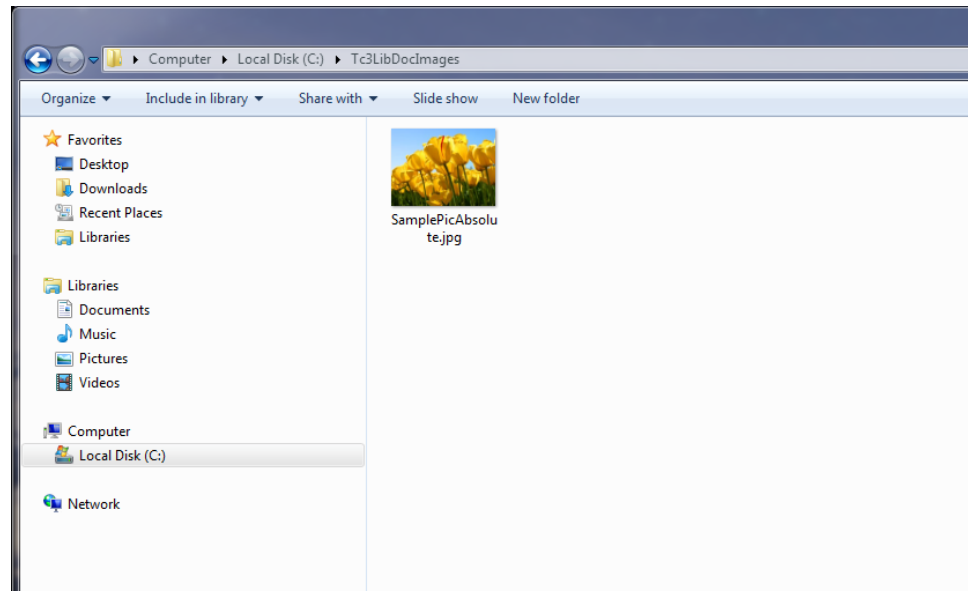
Specify an absolute file path if you have saved the image in an arbitrary folder in your file system. Specify a relative path if the image is integrated in the library project.

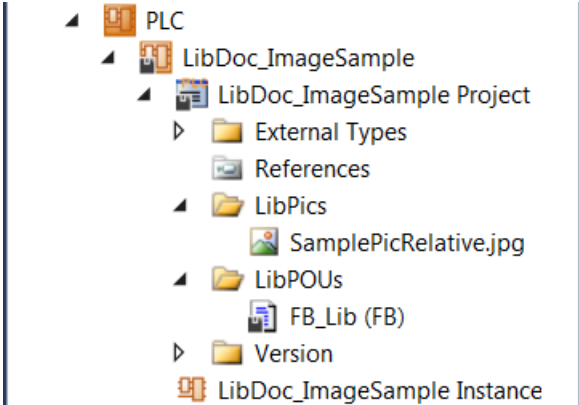
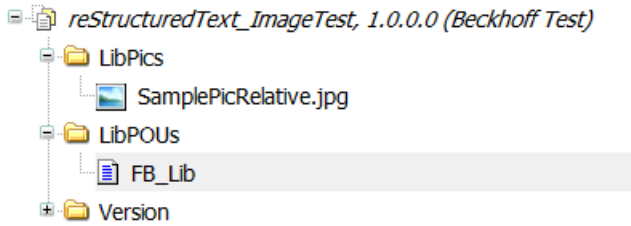
● Forwarding the library file

i If you integrate the images in the library project and use relative file paths and then forward the library project as a file, the required image information will be sent with it directly as part of the library project. If you use absolute file paths, the images must be forwarded separately and must exist at the place in the file system specified in the file path for the person who uses the library. The latter is also a static approach in which the organization and maintenance of the image information in the file system means additional work.

Absolute file path

e.g. *C:\Tc3LibDocImages\SamplePicAbsolute.jpg*



<p>Relative file path Requirement: TwinCAT 3.1.4022.22</p>	<p>e.g. <code>../../LibPics/SamplePicRelative.jpg</code></p> <p>Integrating an image in the PLC project</p> <ol style="list-style-type: none"> 1. Create a folder for images in the PLC project tree for a uniform storage place and clear project structure (e.g. LibPics). 2. Select the command Add > Existing element in the context menu of the folder. 3. In the file system, select the image (e.g. SamplePicRelative) and confirm the dialog. <ul style="list-style-type: none"> ⇒ The image is added to the PLC project and can be referenced in a reStructuredText comment in the documentation of a library object. (See: Examples [▶ 353]) ⇒ When saving the PLC project as a library the image is saved with it in the library file (*.library/*.compiled-library). If the library is referenced in a PLC project, the image is displayed with it in the library manager. The image can be opened by double-clicking on the image in the library manager. <p>Solution Explorer:</p>  <p>Library Manager:</p> 
---	--

Samples

(In [sample project \[▶ 291\]](#): B_DocuElements\Images\FB_Libdoc_Images)

Absolute or relative file path as directive argument

```
(*
Absolute path
.. image:: C:\Tc3LibDocImages\SamplePicAbsolute.jpg
   :width: 70px

Relative path
.. image:: ../../LibPics/SamplePicRelative.jpg
   :width: 70px
*)
```

Absolute path**Relative path****Left aligned image**

```
(*
.. image:: C:\Tc3LibDocImages\SampleLib1\img11.jpg
   :width: 20%
```

Description of the function block. Description of the function block. Description of the function block.
 Description of the function block. Description of the function block. Description of the function block.
 Description of the function block. Description of the function block. Description of the function block.
 Description of the function block. Description of the function block. Description of the function block.
 *)



Description of the function block. Description of the function block.
 Description of the function block. Description of the function block.
 Description of the function block. Description of the function block.
 Description of the function block. Description of the function block.
 Description of the function block. Description of the function block.
 Description of the function block. Description of the function block.

Floating image, left aligned

```
(*
.. image:: C:\Tc3LibDocImages\SampleLib1\img11.jpg
   :width: 20%
   :align: left
```

Description of the function block. Description of the function block. Description of the function block.
 Description of the function block. Description of the function block. Description of the function block.
 Description of the function block. Description of the function block. Description of the function block.
 Description of the function block. Description of the function block. Description of the function block.
 *)




Description of the function block. Description of the function block. Description of the function block.
 Description of the function block. Description of the function block.
 Description of the function block. Description of the function block.
 Description of the function block. Description of the function block.
 Description of the function block. Description of the function block.

Floating image, right aligned

```
(*
.. image:: C:\Tc3LibDocImages\SampleLib1\img11.jpg
   :width: 20%
   :align: right
```

Description of the function block. Description of the function block. Description of the function block.
 Description of the function block. Description of the function block. Description of the function block.
 Description of the function block. Description of the function block. Description of the function block.
 Description of the function block. Description of the function block. Description of the function block.
 *)

Description of the function block. Description of the function block. Description of the function block. Description of the function block. Description of the function block. Description of the function block. Description of the function block. Description of the function block. Description of the function block. Description of the function block. Description of the function block. Description of the function block. Description of the function block. Description of the function block.	
--	---

Inline images can be defined with an `image` directive in a substitution definition (see [Substitution](#) [▶ 344]).

13.6.2.12 Codeblock

The `code` directive allows to display code areas in the comment. These are shaded gray in the Library Manager and displayed in monospaced font with color-coded syntax.

Description	The directive markup consists of an explicit markup start (".. ") followed by the type of the directive (<code>code</code>) and two colons. (See also: Directives [▶ 302])
Principle	<code>.. code::</code>
Properties	<ul style="list-style-type: none"> A blank line is required between the directive and a preceding text body element (for example, a paragraph with text).

Options

Optionally, a directive block can contain a flat list of code options. The following option is recognized:

Number lines : [start line number]	Each line is preceded by a line number. The optional argument is the number of the first line. Default value is 1. The syntax is not highlighted in color in this case.
------------------------------------	--

Samples

(In [sample project](#) [▶ 291]: B_DocuElements\Code Block\FB_Libdoc_CodeBlock)

Code block with color-coded syntax

```
(*
.. code::

    // Attempts to return the value of a boolean property.

    FUNCTION GetBooleanProperty : BOOL
    VAR_INPUT
        sKey: STRING;
    END_VAR

    // This structure defines a special profile.

    TYPE ST_Profile :
    STRUCT
        nId      : INT := -1;
        sBuffer  : STRING(255) := 'Hello';
```

```

END_STRUCT
END_TYPE
*)

```

```

// Attempts to return the value of a boolean property.

FUNCTION GetBooleanProperty : BOOL
VAR_INPUT
    sKey: STRING;
END_VAR

// This structure defines a special profile.

TYPE ST_Profile :
STRUCT
    nId      : INT := -1;
    sBuffer : STRING(255) := 'Hello';
END_STRUCT
END_TYPE

```

Code block with line numbering

```

(*)
.. code::
:number-lines: 1

// Attempts to return the value of a boolean property.

FUNCTION GetBooleanProperty : BOOL
VAR_INPUT
    sKey: STRING;
END_VAR

// This structure defines a special profile.

TYPE ST_Profile :
STRUCT
    nId      : INT := -1;
    sBuffer : STRING(255) := 'Hello';
END_STRUCT
END_TYPE
*)

```

```

1 // Attempts to return the value of a boolean property.
2
3 FUNCTION GetBooleanProperty : BOOL
4 VAR_INPUT
5     sKey: STRING;
6 END_VAR
7
8 // This structure defines a special profile.
9
10 TYPE ST_Profile :
11 STRUCT
12     nId      : INT := -1;
13     sBuffer : STRING(255) := 'Hello';
14 END_STRUCT
15 END_TYPE

```

13.6.2.13 Internal comments

Internal comments are removed on output and are not displayed in the Library Manager.

Description	An internal comment consists of an explicit markup start (".. ") followed by any indented text. (See also: Explicit markup blocks [► 301])
Principle	<pre>+-----+-----+ ".. " comment +-----+block +-----+-----+</pre>
Properties	<ul style="list-style-type: none"> The comment text can be in the same line as the explicit markup start or indented in the following line. <pre>.. This is a comment .. This is a comment</pre>

Sample

(In [sample project \[► 291\]](#): B_DocuElements\Comments\FB_Libdoc_InternalComments)

```
(*
This is the first paragraph.

.. Comments begin with two dots and one space. Anything can
follow, except for the syntax of footnotes/citations,
hyperlink targets, directives, or substitution definitions.

Since comments are not shown, this paragraph follows the previous paragraph directly.
*)
```

This is the first paragraph.

Since comments are not shown, this paragraph follows the previous paragraph directly.

13.6.2.14 Font style

Inline markup allows words or phrases to be displayed in bold, italic or monospaced font in the Library Manager.

(In the [sample project \[► 291\]](#): B_DocuElements\Font style\FB_Libdoc_FontStyle)

Highlighted text (italic)

Text enclosed by individual star characters is highlighted in italics.

Start and end characters	Start character = end character = "*"
Sample	<pre>This is *emphasized text*.</pre> <p>This is <i>emphasized text</i>.</p>

Highlighted text (bold)

Text enclosed by double stars is highlighted in bold.

Start and end characters	Start character = end character = "**"
Sample	<pre>This is **strong text**.</pre> <p>This is strong text.</p>

Inline literals (monospaced font)

Text enclosed by double quotes is displayed in monospaced font (typescript).

Inline literals can be used for short code ranges or inline code.

Start and end characters	Start character = end character = ""`""
Sample	This text is an example of ``inline literals``. This text is an example of <code>inline literals</code> .

13.7 Other commands and dialogs

13.7.1 Command Add library

Function: The command opens the **Add library** dialog. In this dialog you can add library references in the form of a placeholder to the Library Manager and thus integrate them into your application.

Call: Context menu of the **References** object in the PLC project tree

Prerequisite: The library is installed on the local system in the [Library Repository](#) [► 272].

Placeholder vs. Library

Through the use of library placeholders the adaptation of the library versions used requires very little effort, thus making the process of engineering projects and libraries very flexible. When a library reference is integrated in a project or in a library project, it is advisable to use a placeholder instead of a library. This means that within the PLC project no specific library is integrated, but a placeholder. A "real" library is then assigned to the placeholder.

For more information see: [Library placeholders](#) [► 279].

Due to the advantages of a placeholder, the use of the [Command Add library](#) [► 358] is recommended over the [Add library without placeholder resolution command](#) [► 359].

Add library dialog

The **Add library** command opens a dialog, through which a library reference is integrated in the project as a placeholder with the corresponding placeholder name by default. A prerequisite is that an explicit or implicit placeholder name was issued when the [library was created](#) [► 269] (if no explicit placeholder name is entered when the library is created, the library title is entered as placeholder name by default). If no default placeholder name exists for the library that is to be integrated, which may be the case for older libraries, for example, for which no explicit placeholder name was issued, the library reference is integrated as library.

The library reference added via this dialog is integrated as placeholder in the project; by default the placeholder is resolved as "Always newest version" (" * "), i.e. without fixed version. The version resolution can be adapted via the [Placeholder dialog](#) [► 280] or the [Properties window](#) [► 362].

In the row above the library list, you can search for library names or library function blocks by typing a corresponding character string. By double-clicking on a search result or focusing on a search result + [OK], the placeholder associated with the search result is included in the project or in the Library Manager [► 276].	
Library	List of all the libraries installed in the library repository. You can change the type of listing using the buttons in the top right corner of the dialog: <ul style="list-style-type: none"> • by Library Categories ("Category view"): When sorting by Library Categories, the categories appear as nodes. Clicking on a node opens the list of the corresponding libraries or subcategories. • alphabetically by library title ("List view") You can integrate the desired placeholder into the project or into the Library Manager [► 276] by double-clicking or focusing + [OK].
Company	Library provider

See also:

- [Library placeholders](#) [▶ 279]
- [Library Repository](#) [▶ 272]
- TwinCAT 3 User Interface\ Reference User Interface\ Context menu PLC project
 - Command Install Project Library-
 - Command Install Project Library (unknown versions)
 - Command Update project library folder

Also see about this

- 📖 [Add library without placeholder resolution command](#) [▶ 359]
- 📖 [Library creation](#) [▶ 269]
- 📖 [Library Manager](#) [▶ 276]
- 📖 [Command Properties](#) [▶ 362]
- 📖 [Placeholder](#) [▶ 280]

13.7.2 Add library without placeholder resolution command

Function: The command opens the **Find library** dialog. In this dialog, you can add library references to the Library Manager in the form of a library and thus integrate them into your application.

Call: Context menu of the **References** object in the PLC project tree

Prerequisite: The library is installed on the local system in the [Library Repository](#) [▶ 272].

Placeholder vs. Library

Through the use of library placeholders the adaptation of the library versions used requires very little effort, thus making the process of engineering projects and libraries very flexible. When a library reference is integrated in a project or in a library project, it is advisable to use a placeholder instead of a library. This means that within the PLC project no specific library is integrated, but a placeholder. A "real" library is then assigned to the placeholder.

For more information see: [Library placeholders](#) [▶ 279].

Due to the advantages of a placeholder, the use of the [Command Add library](#) [▶ 358] is recommended over the [Add library without placeholder resolution command](#) [▶ 359].

See also:

- [Command Add library](#) [▶ 358]

Find library dialog

The **Add library without placeholder resolution** command opens a dialog through which a library reference is included in the project as a library and not as a placeholder.

In the row above the library list, you can search for library names or library function blocks by typing a corresponding character string. By double-clicking on a search result or focusing on a search result + [OK], the library associated with the search result is included in the project or in the Library Manager [▶ 276] .	
Company	Filtering the list by manufacturer.
Group by category	<input checked="" type="checkbox"/> : Display libraries in a tree structure grouped into categories. When sorting by Library Categories, the categories appear as nodes. Clicking on a node opens the list of the corresponding libraries or subcategories. <input type="checkbox"/> : Display libraries alphabetically in a flat structure.
Display all versions	<input checked="" type="checkbox"/> : Display all library versions. Version specification "*" means the most recent version available in the repository. <input type="checkbox"/> : Display only the latest library versions. In this display, multiple library selection is possible: hold down the [Shift] key while selecting entries.
Details	Opens the Details [▶ 361] dialog.
Library Repository	Opens the Library Repository [▶ 272] dialog. To add a library, which is not yet installed on the local system, you can use this button to open the library repository for the installation.

See also:

- [Library placeholders \[▶ 279\]](#)
- [Library Repository \[▶ 272\]](#)
- TwinCAT 3 User Interface\ User Interface Reference \ PLC project context menu
 - Command Install Project Library-
 - Command Install Project Library (unknown versions)
 - Command Update project library folder

Also see about this

- 📖 [Library creation \[▶ 269\]](#)
- 📖 [Library Manager \[▶ 276\]](#)
- 📖 [Command Properties \[▶ 362\]](#)
- 📖 [Placeholder \[▶ 280\]](#)
- 📖 [Library creation \[▶ 269\]](#)
- 📖 [Placeholder \[▶ 280\]](#)
- 📖 [Command Properties \[▶ 362\]](#)

13.7.3 Command Try reload library

Function: The command tries to reload the selected library.

Call:

- Context menu of the library in the PLC project tree
- Button in the [Library Manager \[▶ 276\]](#)


Requirement: Loading a library while opening a project failed. The library which failed to load is selected in the PLC project tree or in the editor of the Library Manager.

If, for some reason, a library is not available at the defined repository location when a project is opened, a corresponding error message is issued. Once you have corrected the fault and the library is available again, you can use the command to reload this library without having to exit the project.

13.7.4 Command Delete library

Function: The command removes the selected library from the Library Manager.

Call:

- Context menu of the library in the PLC project tree
- Button in the [Library Manager \[▶ 276\]](#) (symbol: )

Requirement: The library is available in the library manager.


See also:

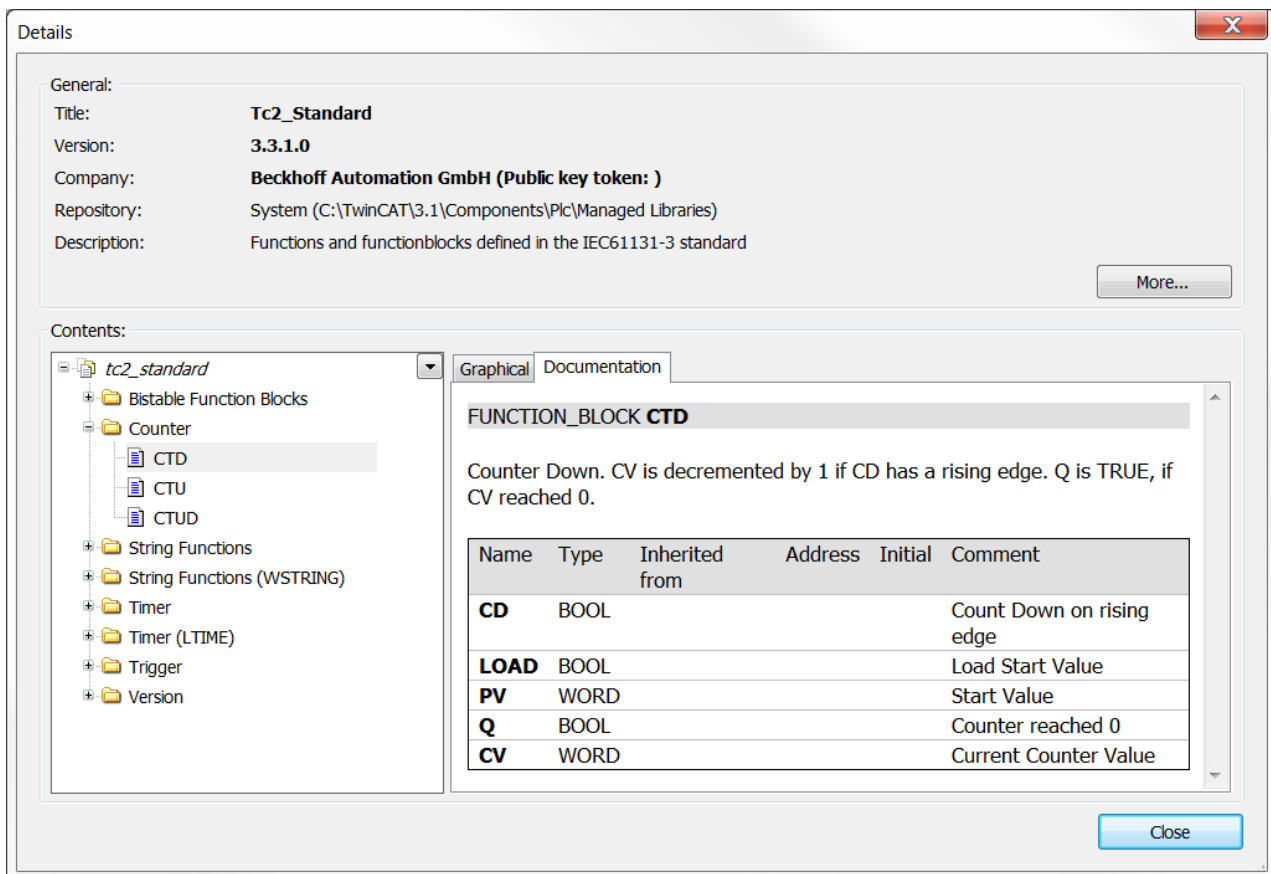
- TwinCAT 3 User Interface\ Reference User Interface\ Context menu PLC project
 - Command Install Project Library-
 - Command Install Project Library (unknown versions)
 - Command Update project library folder

13.7.5 Command Details

Function: The command opens the **Details** dialog with information on the library for the selected version of a library.

Call:

- Context menu of the library in the PLC project tree
- Button in the [Library Repository \[▶ 272\]](#)
- Button in the [Library Manager \[▶ 276\]](#) (symbol: )



Use the **More...** button to obtain the following information:

- Size: specification in bytes
- Created: creation date
- Modified: date of last modification
- Last access: date
- Attributes
- Properties

13.7.6 Command Dependencies

Function: The command opens the **Dependencies** dialog for the selected version of a library, which shows the dependencies of the selected library.

Call:

- Context menu of the library in the PLC project tree
- Button in the [Library Repository](#) [► 272]

If other libraries are included or referenced within the selected library, they are listed in the dialog. For each library reference the title, version and company is shown. References that work with placeholders are represented by the syntax #<PlaceholderName>.

13.7.7 Command Properties

Function: The command activates the **Properties** view for the selected library to configure the settings.

Call:

- Context menu of the library in the PLC project tree
- If the **Properties window** is already open: Select library in the PLC project tree

Properties view

The screenshot shows a window titled 'Properties' with a subtitle 'Tc2_Standard Placeholder Properties'. The window contains a table of properties:

Advanced	
Hide reference	False
Optional	False
Publish all IEC symbols	False
Qualified access only	False
Conditional Referencing	
Condition	
Misc	
Description	Functions and functionblocks defined in the IEC61131-3 standard
Effective Version	3.3.2.0
Name	Tc2_Standard
Namespace	Tc2_Standard
Resolution	Tc2_Standard, * (Beckhoff Automation GmbH) [Default]

Advanced

The following settings are of interest as soon as the library is integrated (referenced) in another library. By default, they are disabled:

Hide reference	<p>TRUE: If the current project is referenced as a library in another project, this library reference is not displayed in the dependency tree and therefore not in the Library Manager either. Thus, you can add "hidden" libraries. It may be difficult to find the cause of compile errors that are due to library errors.</p> <p>FALSE: If the current project is referenced as a library in another project, this library reference is displayed in the dependency tree and therefore not in the Library Manager either.</p>
Optional	<p>TRUE: The selected library is treated as optional. When loading a project that references the library, no error is output even if the library does not exist in the library repository.</p>
Publish all IEC symbols	<p>TRUE: If the current project is later included as a library in another project, the IEC symbols of the selected library reference are published in this project as if the selected library reference were directly included in the project. You can address the library function blocks unambiguously if you use the corresponding namespace path. This consists of the namespace of the "father" library and its own namespace and precedes the function block name.</p> <p>You should only enable this option if you want to create a so-called "container library". A container library is a library that does not define its own function blocks, but only references other libraries to create a kind of library bundle. This can be useful if you have to include several libraries in a project at the same time. In this case it is, however, desirable to place the individual libraries of the top-level package in the Library Manager of the project to shorten the access paths for the library function blocks. This is exactly what you can achieve by enabling this option in advance.</p> <p>FALSE: If the current project is referenced as a library in another project, the function blocks of the library reference displayed in the properties can be addressed unambiguously by using the corresponding namespace. The namespace consists of the namespace of the current library project and the namespace of the selected library reference.</p>
Qualified access only	<p>TRUE: You can access the function blocks and variables of the library in the project only with a namespace prefix.</p>

Conditional referencing

Condition	<p>Here you can add entries that are compared with the compiler definitions set in the PLC project. The library is actively referenced if at least one of the entries corresponds to one of the definitions. If no entry corresponds to any of the definitions, the library is deactivated and displayed grayed out.</p> <p>If you wish to make several entries, you can separate them with a comma.</p> <p>This setting has no effect if no entry exists.</p> <p>Sample: def1,def2,def3</p>
-----------	--

Miscellaneous

Here you can (re-)define which version of the library is used in the project if it is not a library placeholder, or you can edit the placeholder resolution.

Using fixed libraries should be avoided, if possible. Instead, use placeholders to reference the libraries.

Description	Library description
Effective version	Effective version of the library to which the placeholder refers. This field is displayed if the setting "Always newest"/"*" is configured in the Resolution field. This allows the user to see the actual library version used, despite the general "Always newest" resolution setting.
Name	Library name
Namespace	Displays the current namespace. By default, this is identical to the library name unless you have explicitly defined a different default namespace in the project information when you created the library. You can change the namespace for the local project here in the "Properties" view.
Resolution	<p>If you selected a library placeholder in the Library Manager, this field contains the name and version of the library that is to replace the placeholder. At this point, the placeholder is thus resolved as a "real" library and is resolved either as a specific version of the library (e. g. "1.0.0.0") or as "Always newest"/"*".</p> <p>If you want a specific version of the library, select it from the list and TwinCAT uses exactly this version.</p> <p>If you want the "Always newest"/"*" version of the library, select this from the list and TwinCAT always uses the latest version of the library found in the library repository. This may cause the library modules that are actually used to change when a newer version of the library is available.</p>

See also:

- [Using libraries](#) [▶ 266]
- [Placeholder](#) [▶ 280]

13.7.8 Command Set to Effective Version

Function: Depending on whether the **References** object is selected in the PLC project tree or a single library, the command sets all placeholders and libraries available in the project or the selected library or placeholder to an effective version.

Call:

- Context menu of the **References** object in the PLC project tree for setting all placeholders and libraries available in the project to an effective version
- Context menu of the library in the PLC project tree to set the selected library or placeholder to an effective version



If you apply the command to all placeholders and libraries available in the project, the internally used libraries that are not listed in the references will also be set to an effective version.

Example: For example, if a library was previously used as "Always newest"/"*" and the library version 3.3.10.0 was used, the library is no longer referenced as "always newest" by the command **Set to Effective Version** but as fixed version 3.3.10.0.

13.7.9 Command Set to Always Newest Version

Function: Depending on whether the **References** object is selected in the PLC project tree or a single library, the command sets all placeholders and libraries available in the project or the selected library or placeholder to "Always newest" version ("*"). This means that TwinCAT always uses the latest version of the corresponding library found in the library repository.

Call:

- Context menu of the **References** object in the PLC project tree for setting all placeholders and libraries available in the project to “Always newest” version
- Context menu of the library in the PLC project tree to set the selected library or placeholder to “Always newest” version



If you apply the command to all placeholders and libraries available in the project, the internally used libraries that are not listed in the references will also be set to the newest version (*).

Example: In a project, the library “Lib1” is used as “always newest” (“*”). In the library repository that uses development host A, library versions 3.0.0.0 and 3.1.2.0 are installed. The library repository of development computer B has the library version 3.0.1.0 installed.

If the described project is open on development computer A, version 3.1.2.0 of “Lib1” is referenced. If the same project is used on development computer B, library version 3.0.1.0 is used.

14 Multi-task data access synchronization in the PLC

When one set of data is accessed by multiple tasks, the tasks may access the same data simultaneously, depending on the task/real-time configuration. If the data is written by at least one of the tasks, the data may have an inconsistent state during or after a change. To prevent this, all concurrent accesses must be synchronized so that only one task can access the shared data at a time.

These concurrent accesses from several tasks that require synchronization include the following cases, for example:

- direct access to global or other non-temporary variables, for example using operators
- indirect access to global or other non-temporary variables, for example within functions, methods, or other POU calls (especially if a function block instance is globally instantiated)

In brief: If one set of data is accessed by several tasks and the data is written by at least one of these accesses, all read and write accesses must be synchronized. This applies regardless of whether the tasks are run on a single or multiple CPU cores.

⚠ WARNING

Risk of injury due to unforeseen axis movement

If concurrent accesses are not synchronized, there is a risk that the data records will be inconsistent or invalid. Depending on how the data is used in the further course of the program, this can result in incorrect program behavior, undesired axis movement, or even a sudden program standstill. Damage to equipment and workpieces may occur, or people's health and lives may be endangered, depending on the controlled system.

- Synchronize concurrent accesses.
- You will find function tests with corresponding explanations in the sample programs for the MUTEX procedures, which demonstrate why it is necessary to synchronize accesses.

Synchronization options

The following options are available for synchronizing accesses:

- Mutex procedure (TestAndSet, FB_l ecCriticalSection) for securing critical sections [▶ 367]
 - The number of critical sections must always be kept as small as possible.
 - The critical sections must be kept short.
 - For comparatively short critical sections, the use of FB_l ecCriticalSection [▶ 372] is usually recommended.
 - For comparatively long critical sections, the use of TestAndSet [▶ 370] is usually recommended.
- Data exchange via the PLC process image [▶ 373]
 - This variant is only possible and recommended if only one task has write access to the same data.
 - The possible data volume is limited due to necessary internal copy actions. For further information, please refer to the description under "Data exchange via the PLC process image".
- Data exchange via synchronized buffers [▶ 374]
 - This variant is only possible if only one task has write access to the same data.
 - This is a user-specific implementation for which there are various options.
 - Access to the individual buffers must be secured, for example with TestAndSet().
 - The possible data volume is limited due to executed copy actions. For further information, please refer to the description under "Data exchange via synchronized buffers".

Synchronization also in the case of atomic access

The necessity for synchronization normally also applies even if a single access to a variable (e.g. writing an integer) could be described as atomic, i.e. uninterruptible.

Because the property of the atomic access depends among other things on the processor architecture used, every access should be regarded as non-atomic for simplicity's sake and for safety.

It should also be noted that even supposedly safe accesses almost always turn out to be unsafe when considered more closely. This is explained below with the help of two example scenarios:

- As a rule, more than one atomic access is necessary for the desired function expression (e.g. read, change, write). If such a multi-part function expression exists in several tasks, the simultaneous execution on several tasks can lead to a different result than the sequential execution of the expressions.
 - Example: A global counter variable (initialized with 0) is to be incremented by 1 (from two tasks) (`nGlobal := nGlobal + 1;`). If the incrementation is executed at the same time, $0 + 1 = 1$ is calculated both times and the resulting value of the global variable is 1, although the value 2 was intended.
- Several read accesses at different times within a task execution can deliver different variable values.
 - Example: A global variable is written from a task. Its value was previously 50 and is now written to 0 (`nGlobal := 0;`). In a different task context the value of the global variable is queried (`IF nGlobal > 10 AND nGlobal < 20 THEN`). The query consists of two read accesses. If the above write access takes place from the other task between these read accesses, then the condition is fulfilled, even though the global variable did not have a value of between 10 and 20 at any point in time.

Additional Notes

- System operators such as `ADD`, `SIZEOF` or `__NEW` can be called by several tasks simultaneously. This statement only refers to the operator used. If data is accessed during the calls, which in turn is used by several tasks, these data accesses must be synchronized accordingly.
- A function block instance that uses ADS internally may not be used in different tasks. If the instance is nevertheless called from another task, the error `ADSERR_DEVICE_INVALIDCONTEXT=0x709=1801` is issued.
- Due to the execution on the stack, the use of the `STRING` functions (Tc2_Standard library) in different tasks is not critical in TwinCAT 3 (in TwinCAT 2 the `STRING` functions are not safe by default for task changes).
- Also note the possibilities of the compiler extension (documentation [TE1200 TC3 PLC Static Analysis](#)), which provides rules on the topic of “concurrency/competing accesses”. This should be seen as a tool to identify potential synchronization requirements.

14.1 Mutex procedure (TestAndSet, FB_!ecCriticalSection) for securing critical sections

When using mutex procedures or implementing a mutual exclusion, the sections in which competing accesses occur are defined as critical sections. These sections can be synchronized using the function [TestAndSet\(\)](#) [▶ 370] or the function block [FB_!ecCriticalSection](#) [▶ 372] (both from the PLC library `Tc2_System`), so that the sections are placed under mutual exclusion and only one task can access the shared data at a time.

Entering a critical section may depend on one or more conditions. In addition, different critical sections can depend on different conditions.

Examples

- If section 1a reads and section 1b writes to “Data1”, sections 1a and 1b must be synchronized with each other. If sections 2a, 2b and 2c each have read and write access to the “Data2” data, sections 2a, 2b and 2c must be synchronized with each other. The ranges 1x and 2x each depend only on a condition (“Data1” or “Data2” not locked). In addition, in this case they are not interdependent, since different data is accessed in each section. This means that even if section 1b locks the “Data1” data, section 2a can simultaneously lock and access the “Data2” data.

Therefore, the data resources must be synchronized between the following critical sections:

- the use of resource “Data1” between sections 1a and 1b
- the use of resource “Data2” between sections 2a, 2b and 2c

- The situation changes if sections 1x and 2x additionally access the “DataA” data (at least one write access takes place).
Then all sections 1x and 2x are dependent on two conditions: To enter section 1x, data “Data1” and “DataA” have to be enabled, to enter section 2x, data “Data2” and “DataA” have to be enabled. In addition, sections 1x and 2x are now interdependent due to the shared use of the “DataA” data and may no longer be executed simultaneously.
Therefore, the data resources must be synchronized between the following critical sections:
 - the use of resource “Data1” between sections 1a and 1b
 - the use of resource “Data2” between sections 2a, 2b and 2c
 - the use of the “DataA” resource between all sections (1a, 1b, 2a, 2b, 2c)

With the TestAndSet() function, a flag (Boolean variable) represents a condition on which entering the critical section depends (e.g. bLockData1). With FB_!ecCriticalSection, one instance of the function block is used as a lock condition.

Blockage

- TestAndSet(): The function can be used to select and check the content of a critical section. However, the function does not have a blocking effect, and it is possible that the section cannot be processed in a cycle.
- FB_!ecCriticalSection: If another task tries to access an occupied critical section by calling the Enter() method, it is blocked by the TwinCAT scheduler. **The task is blocked until the section is enabled again.**

⚠ CAUTION

Cycle timeout due to stopped task

The duration of the task blocking can lead to the task exceeding the cycle time, depending on the (utilization of the) cycle time.

- Ensure that the critical sections are kept short, in order to avoid cycle overruns in the waiting task. If several tasks are waiting to enter the critical section, access is granted based on their priority.

This results in the advantage of the TestAndSet() function, compared to the function block FB_!ecCriticalSection, that a task is not blocked under any circumstances. The disadvantage is that, in the event that the function TestAndSet() does not grant access, an alternative implementation must be available. This could, for example, be realized via a state machine in order to try access again in the next cycle.

Deadlock

Note that if the function block FB_!ecCriticalSection is used unfavorably, the tasks may become locked, deadlocks may occur. A deadlock always involves at least two tasks. It occurs when the tasks are waiting for each other to release another resource that the other task has already locked.

⚠ CAUTION

Permanent task standstill due to deadlock

Once a deadlock has occurred as described, it can no longer be eliminated programmatically. The tasks involved come to a permanent standstill.

Example:

- Tasks 1 and 2 both require access to the “DataA” and “DataB” data.
- Task 1 initially locks the resource “DataA” and task 2 simultaneously locks the resource “DataB”.
- Task 1 then wants to lock the “DataB” data. Since this resource is already locked by Task 2, this is not possible and Task 1 is locked.
- Task 2, in turn, attempts to lock the “DataA” in addition to “DataB”. Since this is already locked by Task 1, this locking is also not possible and Task 2 is also locked.
- Thus, both tasks are locked. The tasks then waiting for the release of data that the other task locks but cannot release due to its own locking. As a result, the two tasks wait indefinitely for each other.

Even with incorrect synchronization, a deadlock situation is not inevitable. A deadlock only occurs if the processes occur randomly in an unfavorable sequence.

Avoiding deadlocks

In general, you can avoid deadlocks if each task only wants to lock one resource at a time.

You can also avoid deadlocks by only ever requesting and locking resources in a certain sequence.

Example:

- The problem with the above example is that Task 1 and Task 2 request and lock the “DataA” and “DataB” resources in a different order.
- However, if both tasks always lock the resources in the same order, the deadlocking problem is avoided. This means: If each task that requires access to “DataA” and “DataB” in a critical section always requests “DataA” first and, if possible, locks it and only requests and, if possible, locks the “DataB” resource once “DataA” is successfully locked, there can be no deadlock where the tasks “snatch away” the required resources in different sequences as described above.

Sample program: Access synchronization using TestAndSet()

This example shows how shared data can be accessed securely from different task contexts. The data is combined in a structure instance. This also contains a Boolean variable bLocked as a test flag.

Before you read or write to data of this global structure instance, you must request access to it by calling the function `TestAndSet()` [▶ 370] with the corresponding test flag. If access is not granted, you cannot access the data in this cycle. In this case, an alternative treatment must be provided for. If necessary, access is requested again in the next cycle. If access is granted and `TestAndSet()` was called successfully, you can read or modify the data. As soon as you have finished editing, release the access again by setting the test flag to FALSE.

Function test

Start the sample program. Within MAIN1 and MAIN2 there is a counter variable (nLocalBlockedCounter), which increments if the `TestAndSet()` call fails. If the counter is 0, the data access to the global variables was successfully executed on each attempt.

To get a feel for the need for access synchronization, you can run the two tasks from different CPU cores. If you now start the sample program, you will see how the counter variables are incremented irregularly, indicating that data access was occasionally blocked.

Access synchronization is not only necessary for multi-core use, but also when the two tasks run on the same CPU. Mutual task interruptions can also lead to critical inconsistencies in unsaved data access.

Download: https://infosys.beckhoff.com/content/1033/tc3_plc_intro/Resources/7644032779/.zip.

Sample program: Access synchronization using FB_!ecCriticalSection



Windows CE

The functionality of the `FB_!ecCriticalSection` is supported under Windows CE operating systems from TwinCAT v3.1.4022.29 onwards.

The sample shows the use of Critical Sections in the PLC using money transfers for cash accounts. An account is represented by a function block `FB_Account`. Four accounts are involved. All four function block instances are declared in a global variable list to enable access (here: money transfer) from different task contexts.

Each account has an initial balance of 1000. The balance of each account can be read and reset using the `Get()` and `Set()` methods. The following money transfers are implemented in four different task contexts.

Task 1: A->B 500

Task 2: B->C 250, B->D 250

Task 3: C->A 250

Task 4: E->A 250

Once these money transfers have been completed, each account should have a balance of 1000.

It must be ensured that access to an account never occurs from two task contexts at the same time.

The function block `FB_lccriticalSection` [► 372] is used in the `FB_Account`. The method `FB_Account.Lock()` executes `Enter()` on the Critical Section, the method `FB_Account.Unlock()` executes `Leave()` on the Critical Section. Before an account balance can be accessed, the `Lock()` method must be executed successfully. Access to this account is then blocked for others.

To avoid a deadlock, a locking sequence is defined. This should be defined as follows:

Locking sequence: A before B before C before D

This results in the following unlocking sequence: D before C before B before A

Implementation of money transfer within Task 3:

```
(* Task 3: C->A 250*)
IF GVL.fbDepotA.Lock() THEN
  IF GVL.fbDepotC.Lock() THEN
    GVL.fbDepotC.Set (GVL.fbDepotC.Get() - 250);
    GVL.fbDepotA.Set (GVL.fbDepotA.Get() + 250);
    GVL.fbDepotC.Unlock();
  END_IF
  GVL.fbDepotA.Unlock();
END_IF
```

Function test

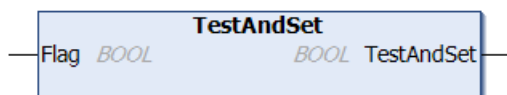
Start the sample program. Within `Main1` you can see the sum of all account balances, which must always be 4000 in PLC online view.

To get a feel for the need to use Critical Sections in this example, you can run the four tasks from different CPUs and set the global variable `blgnoreLock` to `TRUE`. If you now start the sample program, you will see how the account balances assume incorrect values and the sum of all account balances also indicates a money transfer malfunction.

Access synchronization is not only necessary for multi-core use, but also when the four tasks run on the same CPU core. Mutual task interruptions can also lead to critical inconsistencies in unsaved data access.

Download: https://infosys.beckhoff.com/content/1033/tc3_plc_intro/Resources/7643978251/.zip

14.1.1 TestAndSet



You can use this function to check and set a flag. There is no option to interrupt the process. This allows data accesses to be synchronized. The mode of operation of a semaphore can be achieved with `TestAndSet`.

If the function call is successful, the function returns `TRUE` and the desired data can be accessed. If the function call is unsuccessful, the function returns `FALSE` and the desired data cannot be accessed. In this case, an alternative treatment must be provided for.

Inputs/outputs

```
VAR_IN_OUT
  Flag : BOOL; (* Flag to check if TRUE or FALSE *)
END_VAR
```

Name	Type	Description
Flag	BOOL	Boolean flag to be checked <ul style="list-style-type: none"> if it was FALSE, the flag was free and is set (and therefore blocked from now on), and the function returns TRUE if it was TRUE, the flag was already assigned (and therefore blocked), and the function returns FALSE

Sample

```

VAR_GLOBAL
    bGlobalTestFlag : BOOL;
END_VAR

VAR
    nLocalBlockedCounter : DINT;
END_VAR

IF TestAndSet(GVL.bGlobalTestFlag) THEN
    (* bGlobalTestFlag was FALSE, nobody was blocking, NOW
    bGlobalTestFlag is set to TRUE and blocking others *)

    (* ... *)

    (* remove blocking by resetting the flag *)
    GVL.bGlobalTestFlag := FALSE;
ELSE
    (* bGlobalTestFlag was TRUE, somebody is blocking *)
    nLocalBlockedCounter := nLocalBlockedCounter + 1;

    (* ... *)
END_IF
    
```

NEGATIVE sample

Caution is advised with a further encapsulation, e.g. in a function block, as this can destroy the desired atomic operation. Secure synchronization of data accesses can then no longer take place. In the following, a NEGATIVE sample is included that shows how the function may NOT be used. If two contexts were to request access at the same time in this implementation, both might assume that the access is allowed and simultaneous, unsecured accessing of the data would take place.

```

FUNCTION_BLOCK FB_MyGlobalLock
VAR_INPUT
    bLock : BOOL; // set TRUE to lock & set FALSE to unlock
END_VAR
VAR_OUTPUT
    bLocked : BOOL;
END_VAR

IF bLock THEN
    TestAndSet(bLocked);
ELSE // unlock
    bLocked := FALSE;
END_IF

IF NOT GVL.fbGlobalLock.bLocked THEN
    GVL.fbGlobalLock(bLock := TRUE);

    (* ... *)

    GVL.fbGlobalLock(bLock := FALSE);
END_IF
    
```



The function block [FB_IecCriticalSection \[▶ 372\]](#) offers the application of critical sections as an alternative Mutex method.

Requirements

Development environment	Target system type	PLC libraries to include (Category group)
TwinCAT v3.1.0	PC or CX (x86, x64, ARM)	Tc2_System (System)

14.1.2 FB_!ecCriticalSection

The function block is used to make critical sections mutually exclusive. Critical sections are characterized by modifications affecting one or usually several variables, which have an inconsistent state during modifications. It is therefore imperative that such modifications are only carried out by one task at a time. The function block provides the methods Enter() and Leave() for this purpose. A successful call of Enter() makes the critical section accessible; the section is then regarded as occupied. Once the modifications are complete, the critical section must be exited through Leave().

● Cycle timeout due to stopped task

i If another task tries to access an occupied critical area through an Enter() call, it is blocked by the TwinCAT scheduler. The task is blocked until the section is enabled again! Once enabled, processing of the program code continues, and the critical section is entered.

- Ensure that the critical sections are kept short, in order to avoid cycle overruns in the waiting task. If several tasks are waiting to enter the critical section, access is granted based on their priority.

If a task is blocked by the TwinCAT scheduler because it attempted to enter an occupied critical area, this is done without "busy waiting". Low-priority tasks can therefore utilize the CPU capacity during this time.

● Windows CE

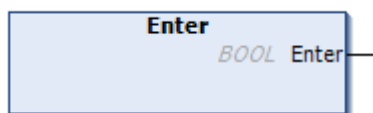
i The functionality is supported under Windows CE operating systems from TwinCAT v3.1.4022.29 onwards. (In older TwinCAT versions the methods return FALSE.)

Alternative

Critical sections can also be realized with the function `TestAndSet()` [► 370]. The function can be used to select and check the content of a critical section. However, the function does not have a blocking effect, and it is possible that the section cannot be processed in a cycle.

As a rule, the number and length of the critical sections should be kept as small as possible.

Enter() method



The method marks the start of a critical section.

Possible return values:

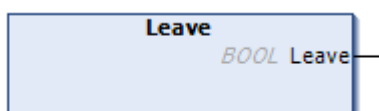
TRUE:

- The critical section may be entered.

FALSE:

- The critical section may not be entered.
- The function block is not yet supported by the runtime.
- The critical section is occupied by another PLC task. This task is on stop in a break point. The return value FALSE avoids permanent blocking of the task and ensures updating of the I/O.

Leave() method



The method marks the end of a critical section. It must always be called when a critical section is completed.

Possible return values:

TRUE:

- The section was exited successfully.

FALSE:

- The function block is not supported by the runtime.
- The critical section was not occupied with Enter.

Application sample for the function block:

The function block FB_IecCriticalSection enables access to shared files to be secured. The instance of the function block and the data to be secured are created globally.

```

VAR_GLOBAL
    fbCrititcalSection : FB_IecCriticalSection;
END_VAR

IF fbCrititcalSection.Enter() THEN
    (* start of critical section *)

    (* end of critical section *)
    fbCrititcalSection.Leave();
END_IF
    
```

Requirements

Development environment	Target platform	PLC libraries to be integrated (category group)
TwinCAT v3.1.4020	PC or CX (x86, x64) WES, WES7, Win7, Win10	Tc2_System (system)
TwinCAT v3.1.4022.29	PC or CX (x86, ARM) WinCE	Tc2_System (system)

14.2 Data exchange via the PLC process image

Data is typically exchanged from the PLC process image with the IO process image. In the same way it is also possible to exchange data within a PLC between two task contexts.

A basic condition for the applicability of this method is that only one task should write to the same data. The data is then declared so that it is part of the output process image of this task.

Example

- “DataA” is to be read from task context 1 and written from task context 2.
- In addition, “DataB” is to be read and written from task context 2 and accessed from task context 1.
- Implementation:
 - You define “DataA” for task context 1 in the output process image and also for task context 2 in the input process image there.
 - Accordingly, you define “DataB” for task context 2 in the output process image and also for task context 1 in the input process image there.
 - You can declare “DataA” and “DataB” directly where they are used, instead of declaring them globally.
 - In the process image display below the PLC instance in TwinCAT XAE, you link “DataA” with each other and “DataB” with each other.

Only the data that is required in the other task context may be added to the output process image, in order to avoid the process image becoming unnecessarily large. The asynchronous mapping performs copy actions in each cycle to exchange the data between the PLC process images and the additional temporary buffer. Because copying actions with large data blocks require a lot of computing time - as is also the case when calling a MEMCPY function - this usually limits the possible data volume to significantly less than 1 MB. The mapping is executed by the respective task itself, so that correspondingly complex mapping reduces the task runtime for the actual program processing.

A second condition for the applicability of this method is that the data must not be function block instances or pointers of any kind (POINTER TO, interface pointers/interface variables, references, ...). It is therefore not possible to call methods from one and the same function block instance from two different task contexts.

If necessary, the required data can be combined into a structure.

Sample program: Synchronization by exchanging data via the PLC process image

The example shows the data exchange between a slow task and a fast task within a PLC instance.

As an example for any data blocks (no function block instances or pointers), an integer variable is transferred to the other task. It also shows how you can trigger a function in the other task context using a Boolean variable.

Download:https://infosys.beckhoff.com/content/1033/tc3_plc_intro/Resources/7643979915/.zip

14.3 Data exchange via synchronized buffers

This synchronization option is a user-specific implementation. As such, no special function blocks of a library are used. The complexity of the program code is greater and also depends on the chosen implementation.

A basic condition for the applicability of this method is that only one task should write to the same data.

Write and read access to the data is synchronized by means of additional temporary data buffers. To synchronize these data buffers, a [MUTEX procedure \[► 367\]](#) is required. However, compared to the exclusive direct use of TestAndSet(), this method has the advantage that access to one of these additional data buffers is always granted and no alternative handling is necessary.

A second condition for the applicability of this method is that the data must not be function block instances or pointers of any kind (POINTER TO, interface pointers/interface variables, references, ...). It is therefore not possible to call methods from one and the same function block instance from two different task contexts.

You may only define data for data exchange that is required in the other task context, in order to avoid the data volume becoming unnecessarily large. The implementation performs copy actions to exchange data between the buffers. Because copying actions with large data blocks require a lot of computing time (calls to the MEMCPY function), this usually limits the possible amount of data to significantly less than 1 MB.

The methods required for this synchronization option are implemented by the user. This specific implementation can be executed in different ways. Some terms such as 3-way buffer or 3-buffer principle are also used for the type of implementation.

Sample program: Synchronization by exchanging data via synchronized buffers

The example shows the data exchange between a slow task and a fast task within a PLC instance.

As an example for any data blocks (no function block instances or pointers), two structures ST_DataA and ST_DataB are transferred to the other task. "DataA" is to be read from task context 1 and written from task context 2. In addition, "DataB" is to be read and written from task context 2 and accessed from task context 1. "DataA" and "DataB" are defined as structure instances for task context 1 and task context 2.

The example contains the function block FB_DataSync with the respective extensions FB_MyDataASync and FB_MyDataBSync for synchronizing these data buffers. Instances of these function blocks are declared in a global variable list and used in the program sequence. Furthermore, only one task context may write to these function block instances – method Write() – and another one may read - method Read(). For security reasons, the methods fail if ignored (return value FALSE).

As mentioned at the beginning, the implementation is more complex than with other PLC examples. In this example, the actual conversion of the data exchange is in function block FB_DataSync, which manages access to several buffers. If you want to test the functionality with your own new data structure, you can leave FB_DataSync unchanged. As with FB_MyDataASync, define a new function block derived from FB_DataSync. In addition, you instantiate your own data structure twice and append the methods Read()/Write(), which each require the assignment of a reference to your new data structure and internally call the methods ReadData()/WriteData() of the base class.

To convert FB_DataSync in this example, two internal buffers are used for temporary data storage. The writing task context copies its data into these buffers and the reading task context copies its data from them. The function TestAndSet() ensures correct access to one of the two internal buffers. This means that the same data is not accessed simultaneously, although data can be written and read simultaneously externally in the application code.

Download:https://infosys.beckhoff.com/content/1033/tc3_plc_intro/Resources/7643976587/.zip

15 Creating a visualization

In contrast to TwinCAT 2, the visualization clients simply act as interpreters of drawing commands. Regardless of its type ([PLC HMI](#) [[▶ 603](#)], [PLC HMI Web](#) [[▶ 608](#)]), each client receives the same instructions, so that the resulting visualization is the same for each type. The consequence of this is that the associated visualization application always has to be downloaded to the PLC. This may represent a problem for small controllers.

A distinction is made between three different visualization variants in the PLC project:

- [Integrated visualization](#) [[▶ 602](#)] – visualization runs in the development environment of TwinCAT on the programming system
- [PLC HMI](#) [[▶ 603](#)] – visualization runs without development environment on the control computer or a third computer
- [PLC HMI Web](#) [[▶ 608](#)] – visualization runs in a browser

Despite the three different variants, engineering is necessary only once, as a result of which the look and feel is identical in all visualizations. PLC HMI and PLC HMI Web can be enabled by simply adding [TargetVisualization](#) [[▶ 606](#)] and [WebVisualization](#) [[▶ 609](#)] objects. The availabilities of the individual [Visualization elements](#) [[▶ 404](#)] and functions in the different variants can be found in the section "[Availability](#) [[▶ 610](#)]"

A visualization can consist of several visualization pages. Each of these pages is managed in a [visualization object](#) [[▶ 402](#)]. In the settings of a visualization object it is possible to change the [size and the intended use](#) [[▶ 403](#)] ("Visualization", "Numpad/Keypad", "Dialog"), for example. When the first visualization page is added, the [visualization libraries](#) [[▶ 401](#)] and a [Visualization Manager](#) [[▶ 387](#)], which is used for managing general settings for all visualization pages of a PLC project, are automatically added.

A visualization page is developed in a [Visualization Editor](#) [[▶ 376](#)]. This editor is supplemented by a [Toolbox](#) [[▶ 384](#)], which provides the available visualization elements, and a [Properties window](#) [[▶ 385](#)] for configuring the added elements. The visualization elements are provided by the visualization libraries, based on the currently selected [profile](#) [[▶ 402](#)]. The visualization elements can be [arranged and grouped](#) [[▶ 377](#)] conveniently as required.

In contrast to TwinCAT 2, all visualization elements and the visualization objects themselves are implemented as IEC61131-3 function blocks. They can be saved in libraries and thus be made available in other projects. Moreover, the visualization pages managed in the visualization objects can be instantiated, i.e. [referenced](#) [[▶ 612](#)], in other visualization pages. This concept is complemented by [placeholders](#) [[▶ 612](#)] in the form of inputs and outputs, through which the referenced pages can be configured. The placeholders can be edited in an interface editor. In this way it is possible to develop default pages or dialogs in a one-time effort for subsequent unlimited reuse with different parameterizations.

The visualization elements can be animated via project variables, which can be entered in the corresponding settings, either directly or via expressions, i.e. combinations of the variables with operators and constants. This enables scaling of variable values, for example, to make them suitable for application in the visualization. Elements of type rectangle can thus be programmed for an absolute movement, for example.

In addition a [user administration](#) [[▶ 393](#)] and a [language selection](#) [[▶ 616](#)], based on text lists, can be implemented. ANSI and [Unicode](#) [[▶ 388](#)] are available as character encoding variants in the visualization. Within a visualization it is also possible to use any expressions, even function calls.

TwinCAT 2 visualizations can be imported.

15.1 Visualization Editor

A visualization can be created and configured with the visualization editor. The editor can be opened by double-clicking on a [Visualization object](#) [[▶ 402](#)] in the PLC project. It features three further editors:

- [Interface editor](#) [[▶ 378](#)] for defining placeholders
- [Hotkeys configuration editor](#) [[▶ 382](#)] for linking actions to keys or hotkeys

- [Element list \[▶ 383\]](#), which contains a list of all elements of the visualization page that is open in the visualization editor

The elements can be shown or hidden via the menu item 'Visualization' or via the arrows at the top of the visualization editor.

The visualization editor is functionally extended by the [toolbox \[▶ 384\]](#), which makes the existing [visualization elements \[▶ 404\]](#) available for inserting in the visualization editor, and the [Properties window \[▶ 385\]](#), in which the properties of the element that is currently highlighted in the visualization editor can be displayed and edited. They can be opened via the menu item "View".

Selecting visualization elements

A [Visualization elements \[▶ 404\]](#) selected in the visualization editor shows small black squares in its frame, which can be selected with a further mouse click and moved. In this way the position and size of the element can be modified.

To select an element, pull the cursor over the element. When the cursor symbol takes the form of a hand. Press the left mouse button. Multiple selection is possible. To this end, press and hold the Shift key and select the individual elements, or draw a rectangle around the elements while pressing and holding the left mouse button. The elements can also be selected in the [element list \[▶ 383\]](#). Individual sub-elements of a group can then also be selected.

Keyboard operation

If an element is selected, the selection can be moved to the next element in the addition sequence via the tab key – or to the previous one with Shift + Tab.

Use the space bar to open a text input field in the currently selected element. Enter a string and press the Enter key to save it.

The properties of a selected element are immediately displayed in the Properties window and can be edited there. If several elements are selected, changes to the properties are applied to all elements.

Position, size, alignment, order

After the insertion of a [visualization element \[▶ 404\]](#), the following properties can be changed directly in the editor window; note that the size and position can also be edited in the [Properties window \[▶ 385\]](#).

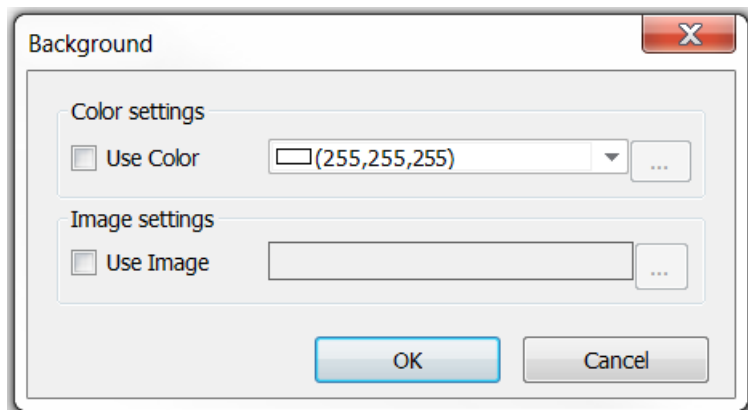
Position	Either select the element with a mouse click and drag it directly to the required position without releasing the mouse button, or click once more on an already selected element and then move it while holding mouse button or via the arrow keys. Another option is to edit the position values in the Element properties [▶ 385] .
Size	Select the element, then click on one of the small squares in its frame. The cursor icon – crossed double arrows or a vertical or horizontal double arrow – indicate in which direction the size of the square can be modified.
Alignment	To align two or more elements with each other, first select the elements you want to align. Then open the submenu "Alignment", either via the menu item "Visualization" or via by right-clicking in the visualization editor, to change the alignment.
Order	To position one or several visualization elements in the background, the foreground or in between in the visualization, first select the element(s) you want to position. Then open the submenu "Order", either via the menu item "Visualization" or via by right-clicking in the visualization editor, to change the order.

Grouping of visualization elements

To consolidate several visualization elements in a group, first select the elements you want to group. Then select "Group", either via the menu item "Visualization" or by right-clicking in the visualization editor, to group the elements. Once a group has been formed, subelements of the group can only be selected via the [element list \[▶ 383\]](#).

Background image

A visualization page offers an option to specify a background color and/or a background image. Right-click in the visualization editor to open a context menu from which the entry "Background" can be selected. This entry opens the following dialog:



There are checkboxes for background color and background image. To select an image as a background image, it first has to be added to an [image pool](#) [► 146] in the PLC project.

15.1.1 Interface editor

The interface editor is used to define interfaces in visualizations. The interfaces are intended for referencing in a frame on another visualization page. The editor is part of the visualization editor and appears in the upper part of the editor as a separate tab next to the editor for the [hotkeys configuration](#) [► 382] and the [element list](#) [► 383].

Interface declaration

Since a visualization is treated as a function block, the interface editor appears as a normal declaration editor in the upper part of the visualization editor. Input variables, such as frame parameters, can be found here.

Parameter declarations

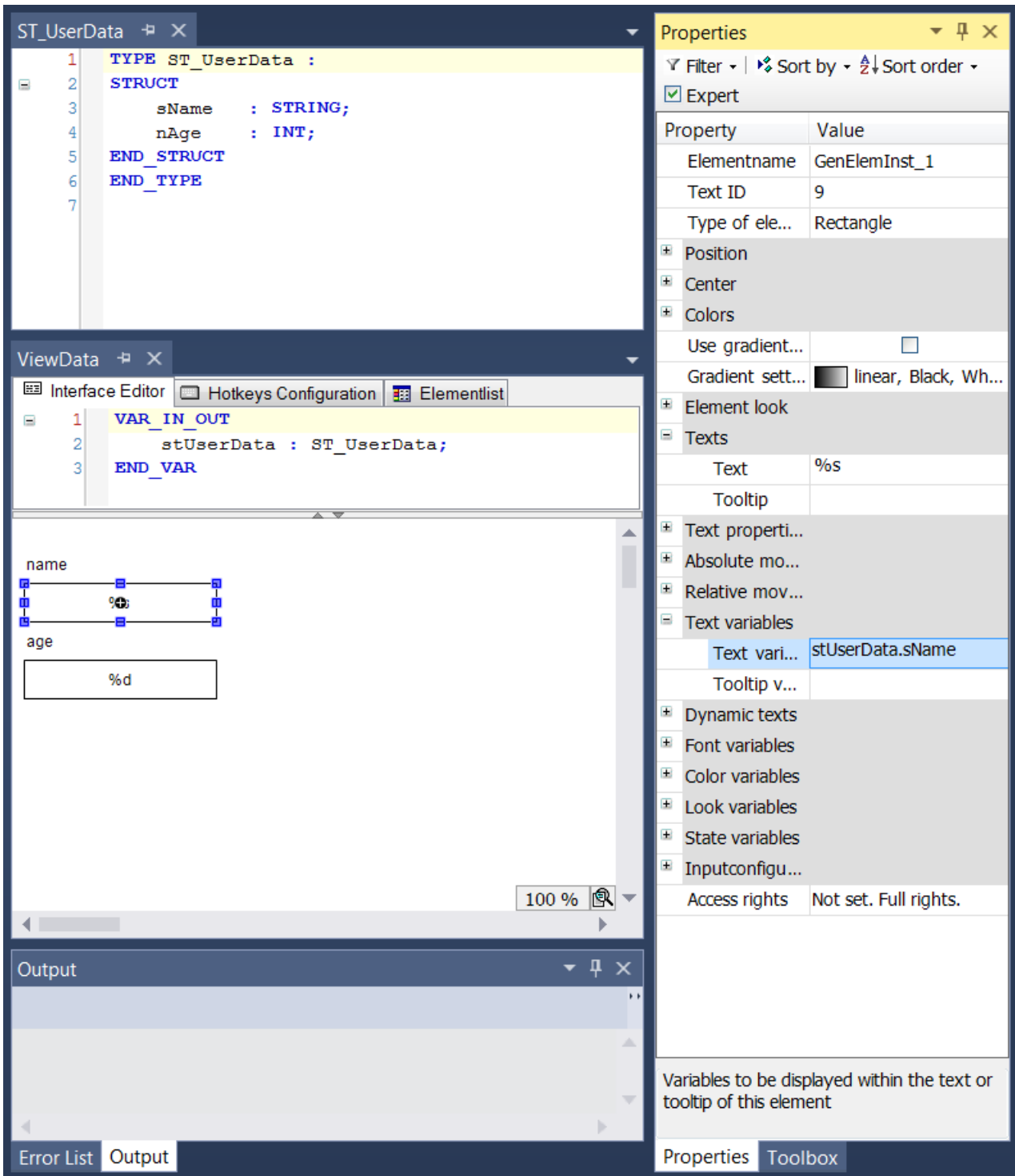
- VAR_INPUT
- VAR_INPUT with attribute 'parameterstringof': A variable with this attribute is given as its value the name of the variable that is set to a to-be-defined input variable when called in a frame element.

```
VAR_INPUT
{attribute 'parameterstringof' := 'bInput'}
sVarName : STRING;
bInput : BOOL;
END_VAR
```

- VAR_IN_OUT: If a data structure is transferred, it has to be defined as VAR_IN_OUT. Without pragma, the parameter values are copied when the visualization is opened. Entries are written to the copied data structure. When the visualization is closed, the values are written back to the parameters.
- VAR_IN_OUT with VAR_IN_OUT_AS_POINTER attribute: Objects are allowed to be transferred as references to visualizations. This can be useful in situations where information is to be transferred to a visualization without actually copying the information, or for updating information from outside, while the dialog is open. This attribute is only permitted for visualizations that are set and used as dialogs.

Example

The ViewData visualization uses the interface stUserData of type ST_UserData. The visualization element GenElemInst_1 of the ViewData visualization is user-based and shows the variable value of stUserData.sName, as programmed in the View properties of GenElemInst_1.



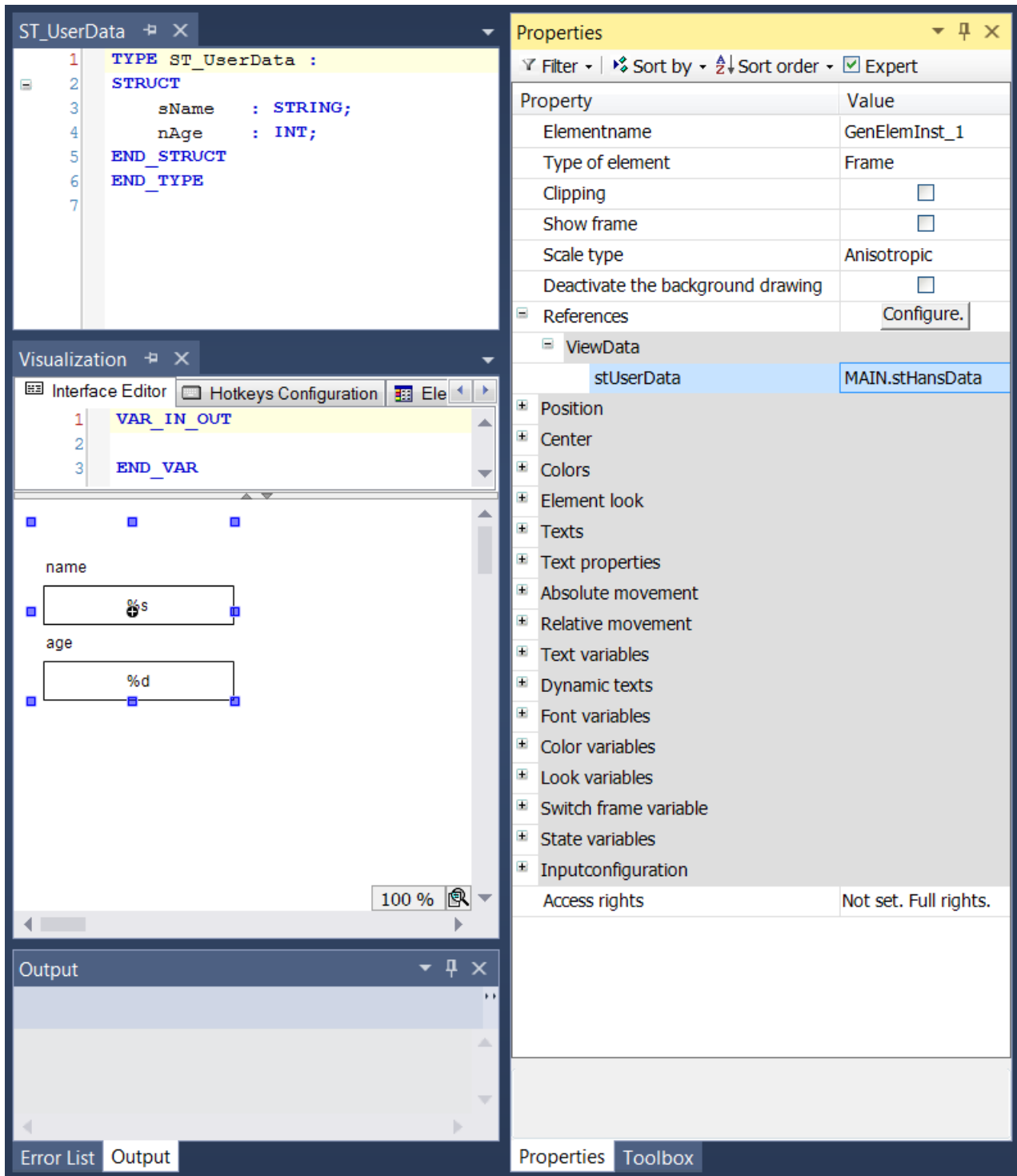
Programming of frame

To program a frame element, proceed as follows:

1. Add a frame element in your main visualization.
2. Set the reference to a visualization with interface, which can be stored in a library.
3. Assign a variable to each frame parameter.

Example

The Frame visualization element references the ViewData visualization. In Properties view, the assignment is listed together with all frame parameters. Click in the value field of a such variable to assign a variable of the same type. MAIN.stHansData is also assigned here.



Updating the frame parameters

If the interface of a visualization that was added to a frame is modified, the dialog shown in the example is opened automatically when the project is saved or compiled, or when an affected object is opened. You can then adjust the frame parameters in this dialog. If the interface of a visualization that originated from a library was modified, the parameters in your project also have to be updated.

Parameter	This column shows the modified parameters in a tree view. First, the visualization is listed, then the corresponding frame with the reference to the modified visualization. Below this, the currently valid parameters are listed, followed by the previous parameters.
Type	This column shows the parameter type.
Value	The value field contains the variable that was assigned to the parameter. Note the color of the field: <ul style="list-style-type: none"> • Beige: This parameter was automatically taken from the previous configuration. • Grey: This new parameter has not yet been assigned a value. • Green: This new parameter has already been assigned a value. • White: There is nothing to configure here.

Edit values

To edit values, select the value field with a mouse click. Click in the field or press the space bar to open a line editor. Or simply start typing, once you have selected the field. To assign a different variable, type its name or select a variable using the input wizard.

An existing value entry can be copied to another field. Select the entry to be copied, use "Copy" and select the field in which you want to insert the entry, then use "Paste".

Confirm the setting in the dialog with "OK". The dialog closes, and the new parameter assignments can be displayed in the References category under Properties view and the corresponding frame element.



Using instance names in the visualization

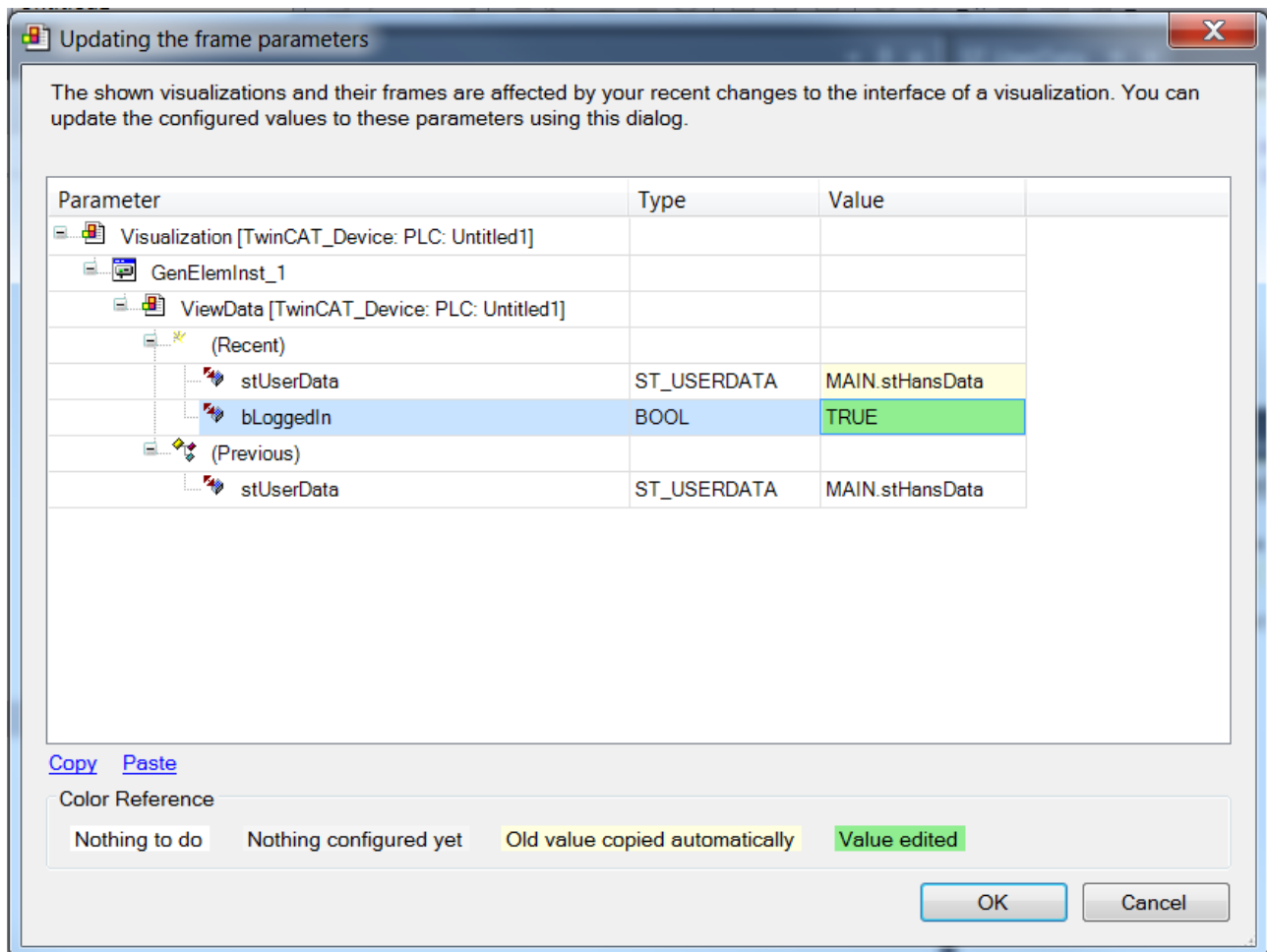
To use an instance name of a transferred variable within the referenced visualization function block, use the pragma attribute '`'parameterstringof [▶ 819]`', which provides a corresponding string.

Example

The interface of the ViewData visualization has been extended with the variable bLoggedIn:

```
VAR_IN_OUT
stUserData : ST_UserData;
bLoggedIn : BOOL;
END_VAR
```

The following dialog for configuring the new parameter appears when a project is compiled or saved:



It contains a comparison between the current and the previous parameters. bLoggedIn was assigned the new value TRUE. Once the dialog has been closed with OK, the new parameters can also be reproduced in the properties of the frame element.

Pragmas

The following two parameters can be used in the interface editor.

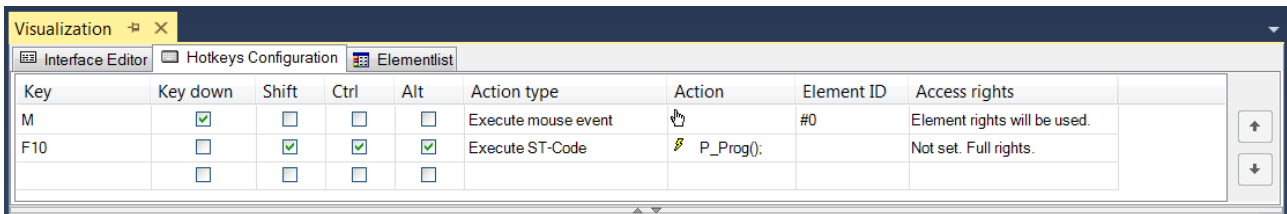
- Attribute "VAR_IN_OUT_AS_POINTER"
- Attribute 'parameterstringof'

15.1.2 Hotkeys configuration

In addition to the [default hotkeys \[▶ 621\]](#), special hotkeys for operating a visualization page in online mode can be defined in this part of the visualization editor. That is, an action can be assigned to a certain key or hotkey. The key configuration only applies to the present visualization page. Key configurations that are to be usable on all visualization pages of the PLC project should be defined in the [Default Hotkeys \[▶ 392\]](#).

In this context please note the "Hotkey" property of a visualization element, which can be set in the [input configurations \[▶ 438\]](#). Such an element-specific key configuration is also managed in the hotkeys configuration editor. It can be edited in both places, with updates being applied in both editors.

The hotkeys configuration editor can be found as a separate tab next to the [interface editor \[▶ 378\]](#) and the [element list \[▶ 383\]](#) in the upper part of the visualization editor.

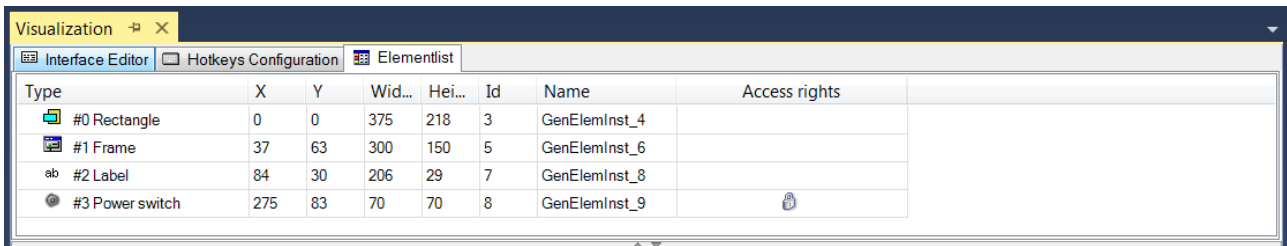


In each row of the table editor, a key or a hotkey can be linked with an action. See the following columns:

Key	Name of the key. The key name can be entered manually or via a selection list, which can be opened by double-clicking on the cell. The list contains all the keys that are defined in the device description.
Key down	If this option is enabled, the action is executed when the key is pressed. Otherwise, it is executed when the key is released. If the action is to be executed both when the key is pressed (KeyDown) and when the key is released (KeyUp), the table has to contain two corresponding definitions for the key.
Shift, Ctrl, Alt	If this option is enabled, the Shift or Ctrl or Alt key has to be pressed together with the key, in order to execute the action.
Action type	The action type can be defined via a selection list, which can be opened by double-clicking on the cell. The action types correspond to the types that are available in the input configuration [▶ 406] of a visualization element.
Action	More precise configuration of the action to be executed. This depends on the action type and corresponds to the mouse action, which can be used in the input configuration [▶ 406] of a visualization element.
Element ID	<p>ID of the visualization element, to which the key is allocated via the "Hotkeys [▶ 438]" property. Unique ID within the current visualization.</p> <p>If a key is linked to several actions, they are executed in the same order (top to bottom) as they are listed here in the configuration table. This configuration can be changed by selecting a key definition (click in the table row) and moving it up or down using the arrow keys to the right of the table.</p> <p>Note the following call sequence for processing key actions:</p> <ol style="list-style-type: none"> 1. Event handler for the application, if enabled (optional), e.g. for detecting events and entries 2. Hotkeys configuration [▶ 392] in the Visualization Manager, which applies to all visualizations in the application 3. Definition of the standard keyboard operation [▶ 621] in online mode 4. Special hotkey configuration of the individual visualizations, which are described here, whereby the main visualizations are viewed before the visualizations referenced in frames.
Access rights	<p>This setting can be used to define a selection of user groups that is allowed to use the respective hotkey. Click to open a dialog for selecting the existing groups. The following two status messages exist:</p> <ul style="list-style-type: none"> • Not set. Full rights. <p>The default message is set, if the hotkey can be used for all groups.</p> <ul style="list-style-type: none"> • Rights are set: limited permissions. <p>The message is set, if the hotkey is blocked for at least one group.</p> <ul style="list-style-type: none"> • Element rights are used <p>If the hotkey is allocated to a visualization element via the "Hotkey [▶ 438]" property, the corresponding element rights are automatically applied.</p>

15.1.3 Element List Editor

This part of the visualization editor shows a list of all elements of the visualization page, which is currently open in the editor.



The elements are listed from top to bottom according to their position on the z-axis of the visualization, i.e. the first row refers to the element that is furthest in the background. The following values are displayed but cannot be edited here:

Type	Element type and icon, as used in the toolbox [▶ 384] , and the element number, which initially results from the order in which elements are added.
X, Y	Position of the top left corner of the element (0, 0 = top left corner of the visualization area)
Width, height	Element dimensions
ID	Internally assigned element ID
Name	Element name, as defined in the element properties [▶ 385]
Access rights	If the behavior of an element is limited for some user groups, this is indicated by a padlock symbol.

Element(s) can be selected by selecting the corresponding table row(s), whereby the selection is always synchronized with the main window of the visualization editor. Note, however, that subelements of a group can only be selected here in the element list.

The position of a selected element on the z-axis, i.e. on the background/foreground axis, can only be changed in the [visualization editor \[▶ 377\]](#) itself.

The commands for undoing edit operations in the element list ([Ctrl] + [Z]) or removing elements (Del) can also be used in the element list.

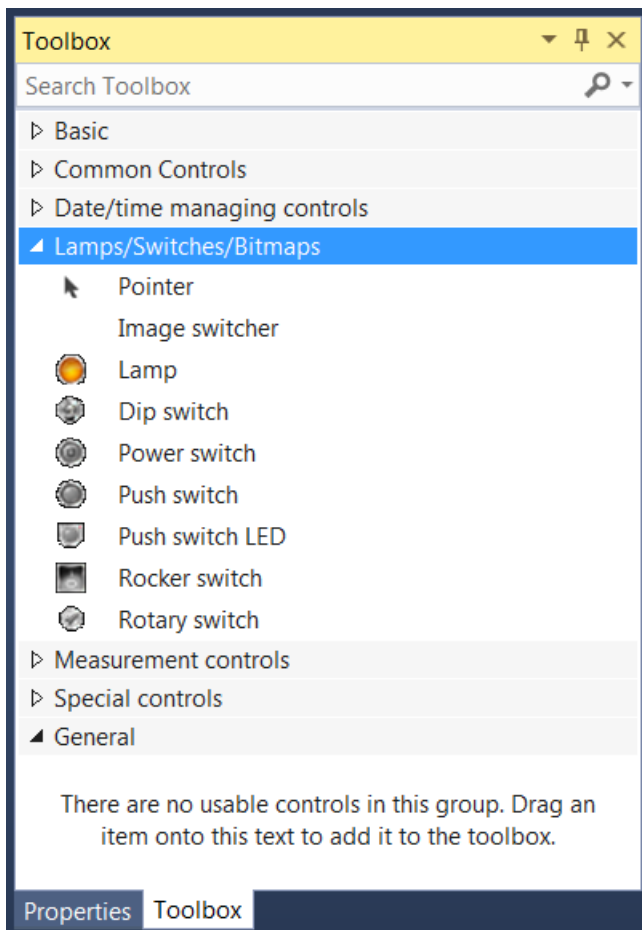
15.1.4 Toolbox

The toolbox makes all the visualization elements available, which can be added in the [visualization editor \[▶ 376\]](#). The elements are made available in the project via [libraries \[▶ 401\]](#). The selection in the toolbox window depends on the currently active [visualization profile \[▶ 402\]](#).

If the toolbox window is not yet visible, it can be opened in the "View" menu via "Toolbox". The toolbox contains the following categories:

- [Common Controls \[▶ 421\]](#)
- [Basic \[▶ 475\]](#)
- Date/time managing controls
- [Lamps/Switches/Images \[▶ 527\]](#)
- [Measuring devices \[▶ 537\]](#)
- [Special controls \[▶ 580\]](#)

Under these categories the corresponding [visualization elements \[▶ 404\]](#) are listed with their symbols and names.



Visualization elements can be inserted in the visualization editor by means of drag and drop. A plus sign at the cursor symbol while an element is dragged indicates that the element can be dropped into the Editor window. When the mouse button is released, the element appears in the editor.

An element can also be added in the "Visualization" menu under "Add visualization element". Here, the same range of elements is available as in the toolbox. If necessary, the visualization menu can be customized via the customize dialog (command category "Visualization commands"), as required.

15.1.5 Properties window

Any [visualization element](#) [[▶ 404](#)] of the visualization page currently open in the [visualization editor](#) [[▶ 376](#)] can be [selected](#) [[▶ 377](#)]. Its [position, size, configuration and alignment](#) [[▶ 377](#)] can then be modified directly in the editor (mouse actions or visualization commands). Further parameters are configured in the properties window.

Property	Value
Elementname	GenElemInst_1
Text ID	9
Type of element	Rectangle
Position	
X	1471
Y	278
Width	150
Height	30
Center	
Colors	
Use gradient color	<input checked="" type="checkbox"/>
Gradient setting	axial
Element look	
Texts	
Text	%s
Tooltip	
Text properties	
Absolute movement	
Relative movement	
Text variables	
Text variable	MAIN.sText
Tooltip variable	
Dynamic texts	
Font variables	
Color variables	
Look variables	
State variables	
Inputconfiguration	

The type of this element (rectangle, rounded rectangle or ellipse)


Properties editor

The properties of a visualization element - except [alignment and order \[► 377\]](#) - can all be configured in the [properties editor \[► 385\]](#). By default, this editor opens next to the visualization editor, or it can be opened explicitly via the "Properties" command (which can be found in the View menu as standard).

A property can be modified by editing the field "Value". To this end, an input field, a selection list, a dialog or checkbox that can be activated is provided in this field, depending on the element type. The value field opens

- after a double-click,
- after a single click in a selected field,
- via the space bar, if the field was already selected.


If a variable is assigned,

- simply enter its name.
- Use the  button to open the input assistant for selecting a variable. The Variables category lists all variables that have already been defined in the project.

Working in the list of properties can be made easier with the aid of default, sorting and filter functions.

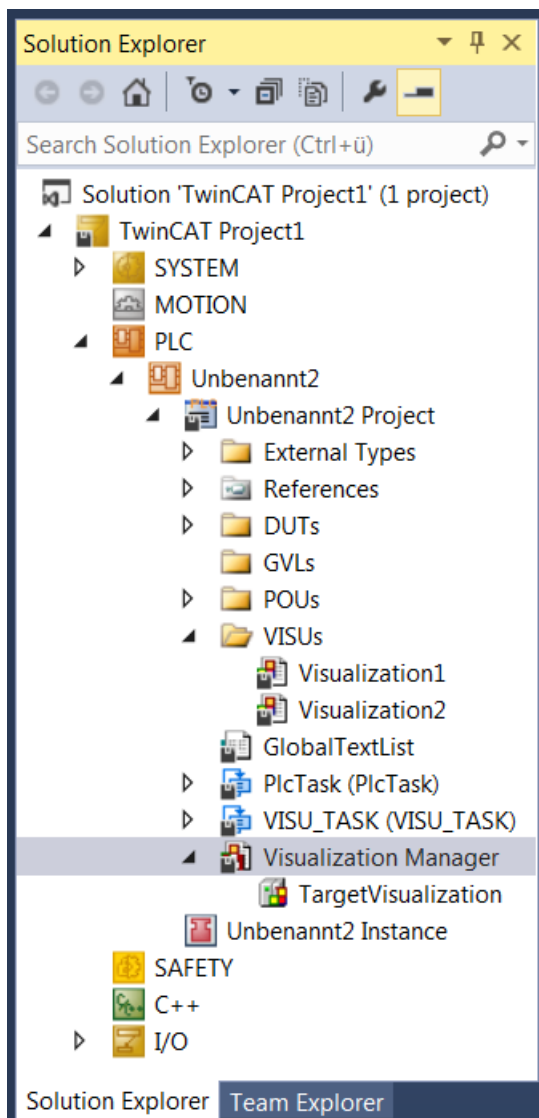
15.2 Visualization Manager

The Visualization Manager manages general settings for all [visualization objects](#) [▶ 402] of a PLC project.

The  Visualization Manager object is added automatically, once the first visualization object was created in the PLC project. An overview of visualization in PLC projects can be found in the section "[Creating a visualization](#) [▶ 376]".

If the function is supported by the device, the client objects for the [PLC HMI](#) [▶ 603] client and/or the [PLC HMI Web](#) [▶ 608] client can be added under the manager. They manage special settings for the respective client type. The [integrated visualization](#) [▶ 602] is used automatically, if no client object was added under the Visualization Manager.

In the example shown in the image below, the Visualization Manager is responsible for the visualization objects "Visualization1" and "Visualization2". In addition, the TargetVisualization object is added, thus enabling the PLC HMI.

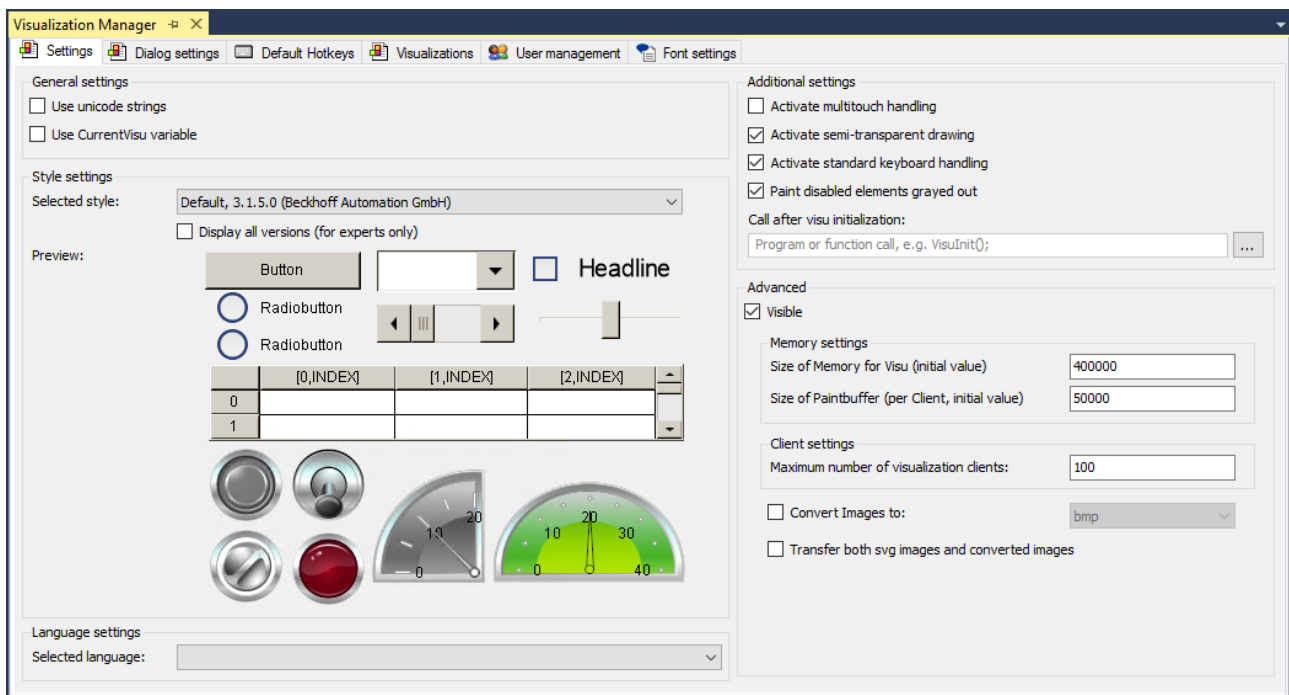


To open the Visualization Manager editor, double-click on the entry. The editor opens in a window, which has the following tabs:

- [Settings \[▶ 388\]](#)
- [Dialog settings \[▶ 390\]](#)
- [Default Hotkeys \[▶ 392\]](#)
- [Visualizations \[▶ 393\]](#)
- [User management \[▶ 393\]](#)
- [Font \[▶ 400\]](#)

15.2.1 Settings

The tab contains settings that apply to all visualizations of the PLC project.



General settings

<p>Use Unicode strings</p>	<p>If this option is enabled, all strings [▶ 614] that are used in the visualization are processed in Unicode format. To do this, "VISU_USEWSTRING" must be entered additionally under Compile > Settings > Compiler defines in the settings for the PLC project [▶ 402].</p>
<p>Use CurrentVisu Variable</p>	<p>The PLC project knows and uses the global variable VisuElems.CurrentVisu of type STRING. It contains the name of the currently active visualization at runtime.</p> <p>The variable can be accessed in read mode to obtain the name of the currently active visualization, and in write mode to change the visualization. The switchover takes place on all display devices in parallel so that the same visualization page is displayed on all connected clients.</p> <p>Requirement: The application contains a visualization that calls further visualizations.</p> <p>Samples:</p> <ul style="list-style-type: none"> • Assigning a variable: <code>VisuElems.CurrentVisu := sVisuName;</code> • Assigning a text: <code>VisuElems.CurrentVisu := 'Visualization1';</code>

Style settings

Selected style	Each visualization presents the elements in this style.
Display all versions (for experts only)	If this setting is enabled, all style versions that are installed on the system are available for selection via the selection menu.
Preview	The elements shown in the preview represent the selected style.

Language settings

Selected language	The selected language is used when a visualization starts.
-------------------	--

i A default language for the start of a visualization can only be set in conjunction with [PLC HMI \[▶ 603\]](#) and/ or [PLC HMI Web \[▶ 608\]](#). With an [integrated visualization \[▶ 602\]](#), the text version 'Standard' is automatically used on startup. [Language change \[▶ 616\]](#) at runtime via buttons on the visualization is also possible in the integrated visualization.

See also:

- [Text and language \[▶ 614\]](#)

i From build 4024.0 the settings for the user management dialogs are listed on the separate [Dialog settings \[▶ 390\]](#) tab and are documented there accordingly.

Additional settings

Activate multitouch handling	At runtime the visualization expects user inputs via gestures and touch events. Affected elements: <ul style="list-style-type: none"> • Elements with input configuration • Elements of type frame • Elements of type tab control
Activate semi-transparent drawing	The visualization draws the elements in a semi-transparent color. When defining the color you can additionally specify a graduation value for the transparency. The transparency is defined in the Transparency property. Preset: Activated. Requirement: You create a new visualization, and the display variants can draw semi-transparently.
Activate default keyboard operation	<ul style="list-style-type: none"> • Tab • Shift + Tab • Input • Up arrow • Down arrow • Right arrow • Left arrow
Draw deactivated elements in gray	The visualization draws deactivated elements grayed out.

Extended settings

Visible	Memory settings, file transfer mode and client settings are visible. (Not required for standard applications)
---------	---

i If the integrated visualization is used, the unavailable settings are grayed out or not displayed at all.

Memory settings

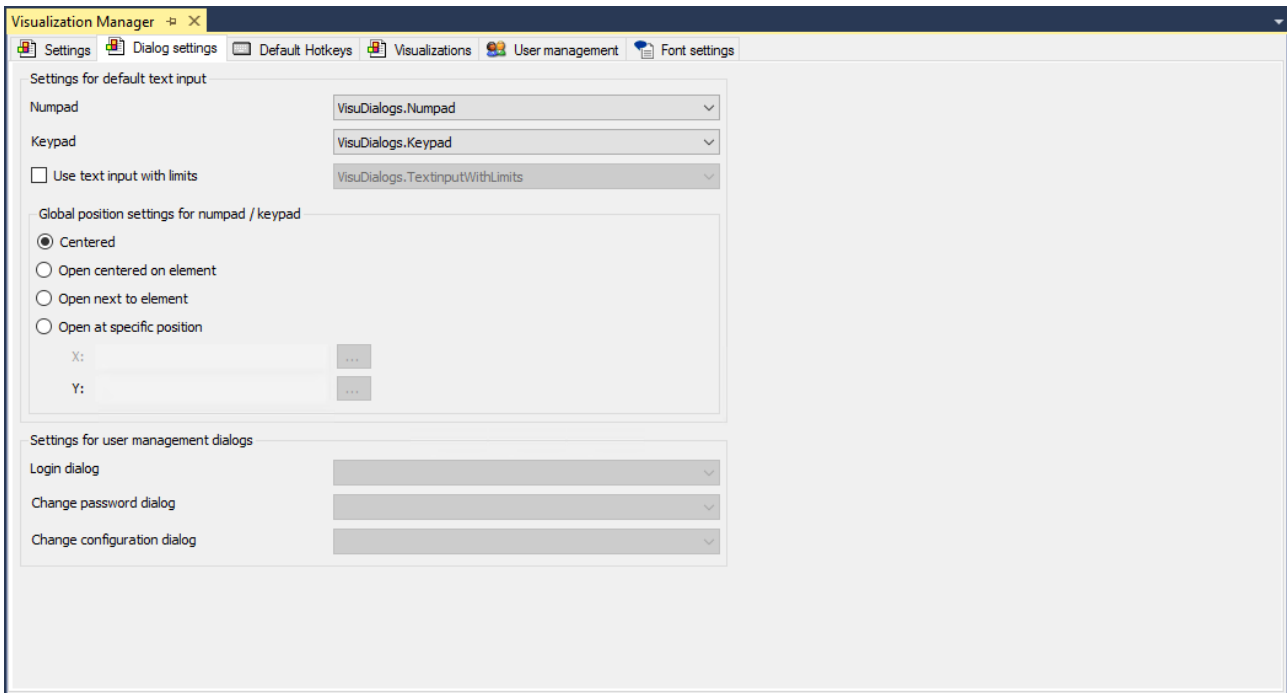
Size of Memory for Visu	Memory size in bytes, which the visualization allocates at runtime. Preset: 400000
Size of Paintbuffer (per Client)	Memory size in bytes, which the visualization allocates for each display option and uses for drawing actions. Preset: 50000

Client settings

Maximum number of Visualization Clients	Limits the number of display variants, which can run simultaneously. If you configure elements such that they vary depending on the display variant, you need to limit the number of display variants. A visualization is assigned an ID at runtime, which identifies the display variant. Data is then processed accordingly. TwinCAT can query the ID with the system variable CURRENTCLIENTID to obtain information on which of the current variants is affected. Sample: <code>arr[CURRENTCLIENTID].dwColor</code> Requirement: Library VisuGlobalClientManager is integrated in the project.
Convert images to	If this setting is enabled, you can select from a drop-down menu whether the images of the standard elements [▶ 404] should be compiled in the bmp or png format. This is necessary if a Windows Embedded Compact operating system is installed on the target system, since only bmp, png and jpg images can be displayed there in the PLC HMI [▶ 603] client.
Transfer both svg images and converted images	If this setting is enabled, image files that had previously been converted to bmp or png can also be loaded to the target system in the original svg format. The PLC HMI [▶ 603] client uses the image files in bmp or png format locally on the Beckhoff CE device and a PLC HMI Web [▶ 608] client uses the image files in svg format. This setting is then available if both the PLC HMI [▶ 603] and the PLC HMI Web [▶ 608] are activated.

15.2.2 Dialog settings

This tab contains default settings for the dialogs that are used in the visualization at runtime for a text input and for the user management. The dialog to be used is defined in the input configuration of the respective visualization element.



The global settings in the Visualization Manager are only effective for use in a [PLC HMI \[▶ 603\]](#) or [PLC HMI Web \[▶ 608\]](#).

Settings for standard text input

A dialog that supports the input appears at runtime for an element with standard text input. You can determine which dialog appears.

Numpad	<p>Dialog in the form of a numpad, which the visualization opens at runtime if the user activates the input field for a number.</p> <p>Preset: VisuDialogs.Numpad</p>
Keypad	<p>Dialog in the form of a keyboard, which the visualization opens at runtime if the user activates the input field for a text.</p> <p>Preset: VisuDialogs.Keypad</p>
Use text input with limits	<p>Requirement: PLC HMI [▶ 603] or PLC HMI Web [▶ 608] is configured as display option. The default text input mode is keypad. In this case the visualization supports text entry via the keyboard at runtime.</p> <p>For inputs with a limited range of values, instead of the input field a dialog can be called up, which shows the range of values.</p> <p>Preset: VisuDialogs.TextinputWithLimits</p> <p>This dialog shows the range of values and does not accept values outside these limits.</p>
Global position settings for numpad / keypad	<ul style="list-style-type: none"> • Centered: The dialog is opened in the center of the screen. • Open centered on element: The dialog is opened on the element and covers it. • Open next to element: The dialog is placed dynamically optimized next to the element. • Open in defined position: The dialog is opened in the visualization window in the position defined here. X, Y: Variable or direct number (pixels) for defining the top left corner of the dialog in the coordinate system of the visualization window. <p>Note: The origin of the visualization coordinate system is the top left corner. The positive horizontal X-axis runs towards the right. The positive vertical Y-axis runs downwards.</p>

Settings for the user management dialog

You can configure your visualization with a user management. To do this you configure an input on an element that causes a user management dialog to appear. In order to use other visualizations as user management dialogs, you have to change the presets here.

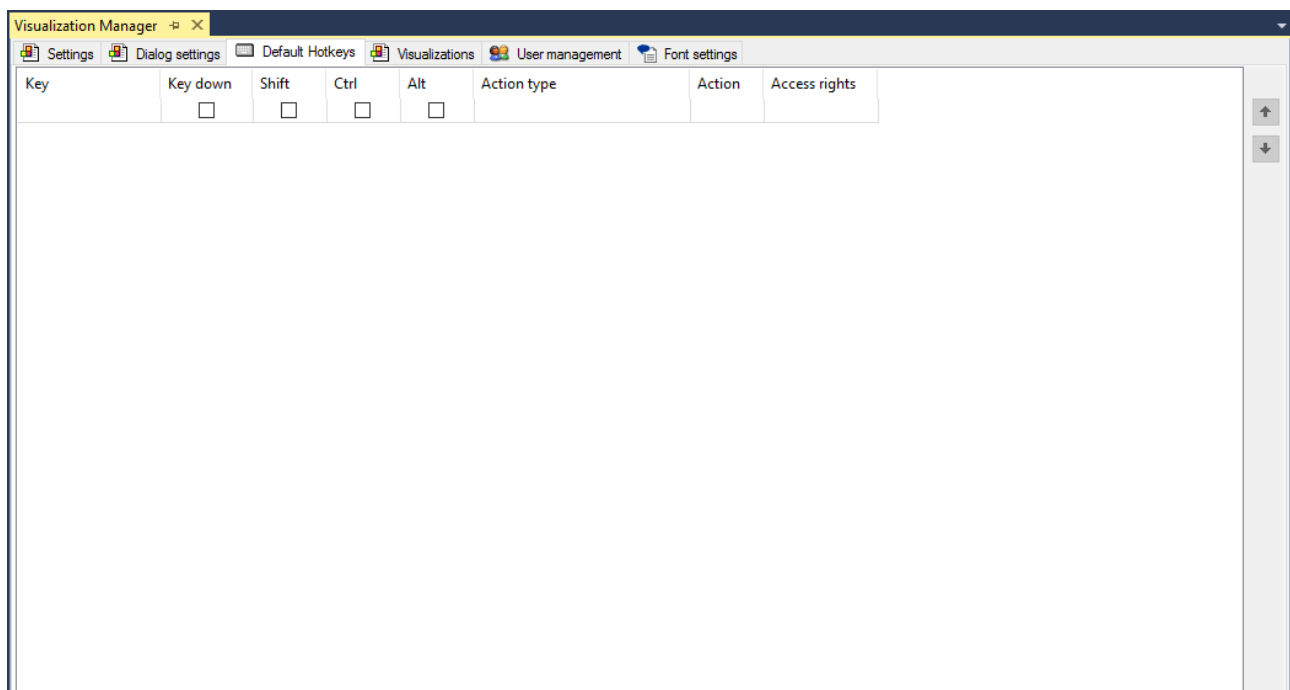
Login dialog	User management dialog that enables a login, typically requiring a user name and password to be entered. The dialog appears upon an input event on an element that, as a follow-up action, executes User management, action Login. Preset: VisuUserManagement.VUM_Login
Change password dialog	User management dialog that enables a password change, typically requiring the current and new password to be entered. The dialog appears upon an input event on an element that, as a follow-up action, executes User management, action Change user password. Preset: VisuUserManagement.ChangePassword
Change configuration dialog	User management dialog that enables a configuration change of the user management, i.e. typically a display of the current user configuration and an option to change it. The dialog appears upon an input event that, as a follow-up action, executes User management, action Open user management. Preset: VisuUserManagement.VUM_UserManagement

See also:

- [User management \[▶ 393\]](#)

15.2.3 Default Hotkeys

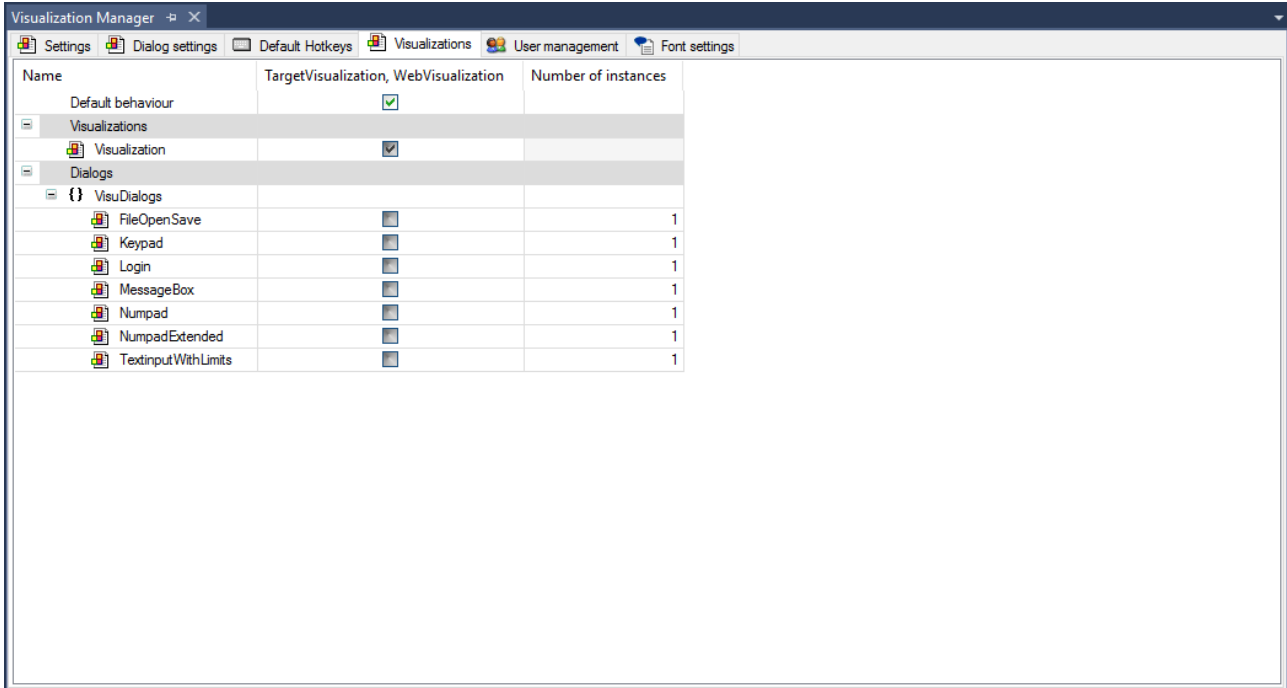
This tab contains the keyboard configuration that applies to all visualization pages in the respective PLC project. In other words, keys / key combinations defined here can be used for user inputs in the visualization in online mode if the respective device supports them.



The configurator is operated in the same way as the one used in the visualization editor for a special visualization. See description of the [Hotkeys configuration editor](#) [▶ 382]. In addition, certain device-independent [default hotkeys](#) [▶ 621] are always available for navigation in a visualization.

15.2.4 Visualizations

This tab lists up all available visualizations and enables an assignment of the visualizations for the loading behavior that depends on the display variants.



Default behavior	<p>If this option is activated the visualization objects in the PLC project are automatically loaded to the respective target system. The activated checkboxes indicate which ones those are.</p> <p>If the option is not activated you can explicitly define the loading behavior for each visualization.</p>
Visualizations	All visualizations created in the project are listed under here.
Dialogs	<p>All visualizations of the type Dialog in the project and from libraries are listed under here.</p> <p>Note: the Number of instances column indicates how often the associated dialog will be instanced.</p>



This functionality replaces the previously possible use of "visualization reference" objects. Such objects can no longer be added, although objects that have already been added will continue to function.

15.2.5 User management

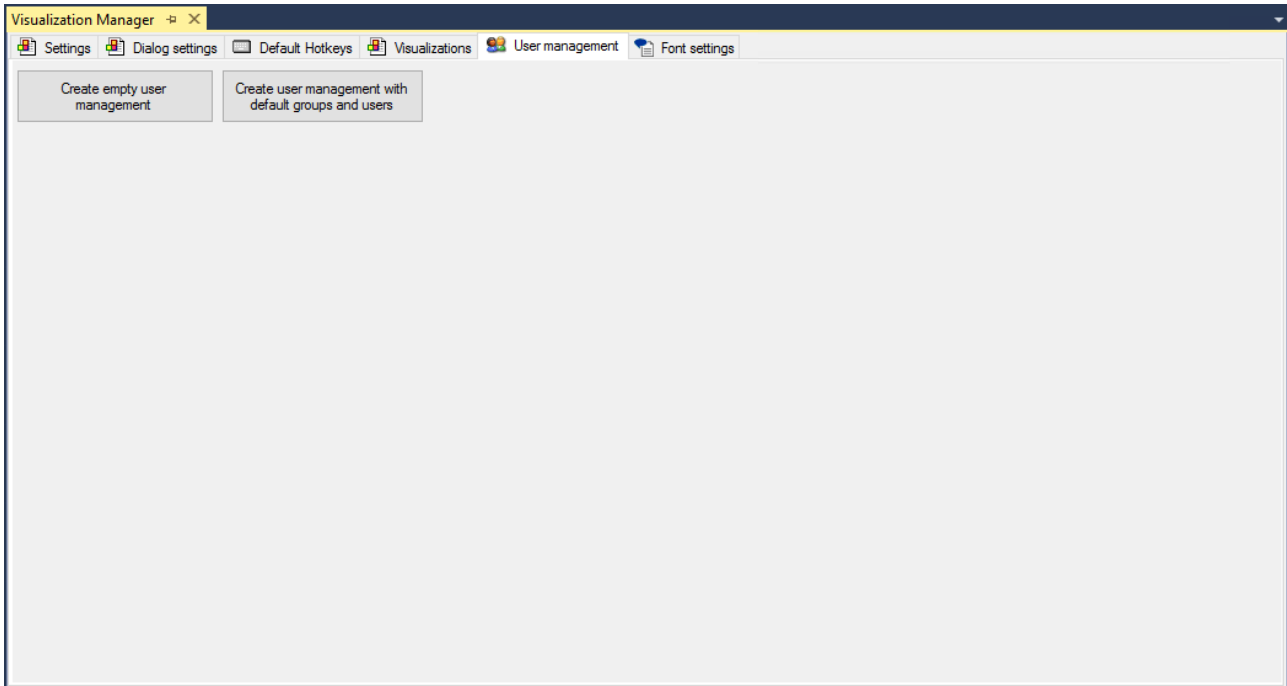
The user management is used to manage the visibility and operability of visualization elements on a user-specific basis. Switching of visualizations can also be configured on a user-specific basis. The users are organized in groups.



The user management can only be used in combination with the [PLC HMI](#) [▶ 603] or the [PLC HMI Web](#) [▶ 608].

First step

First you need to create unique groups and users. Open the "User management" tab in the visualization manager editor. If no user management is configured yet, the following dialog is available:

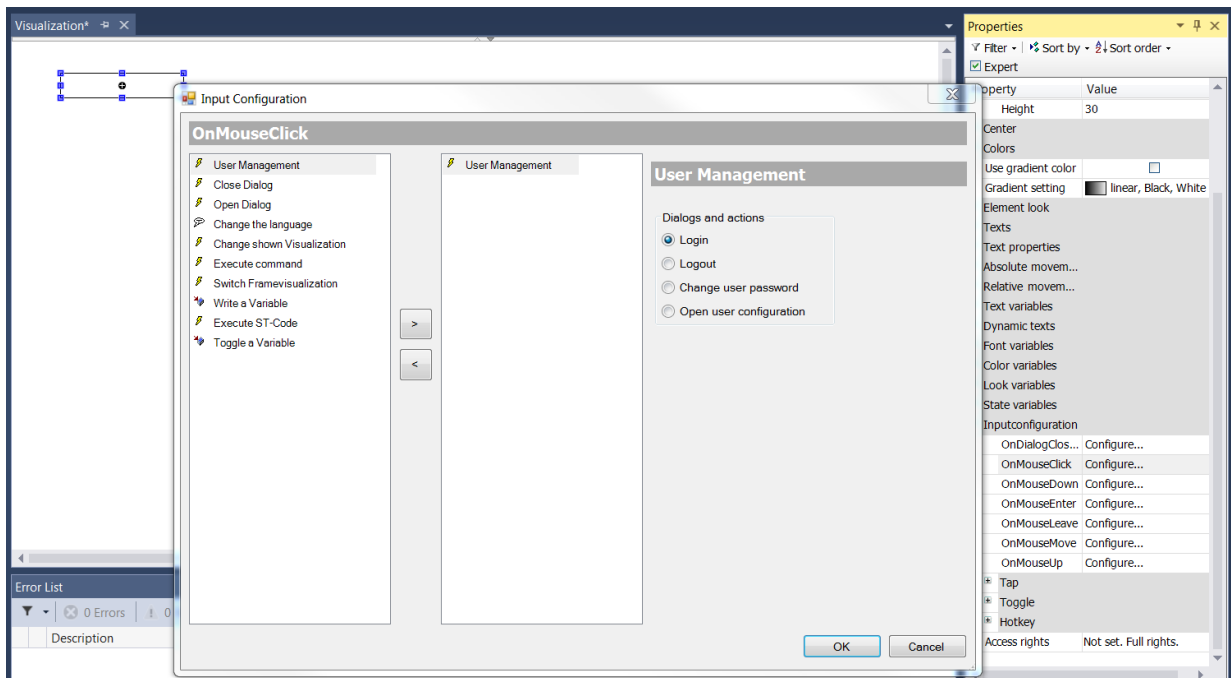


Create empty user management	The user management opens. The group None is created.
Create user management with default groups and users	The user management opens. The following groups and users are created: <ul style="list-style-type: none"> • group Admin with user Admin • group Service with user Service • group Operator with user Operator • group None

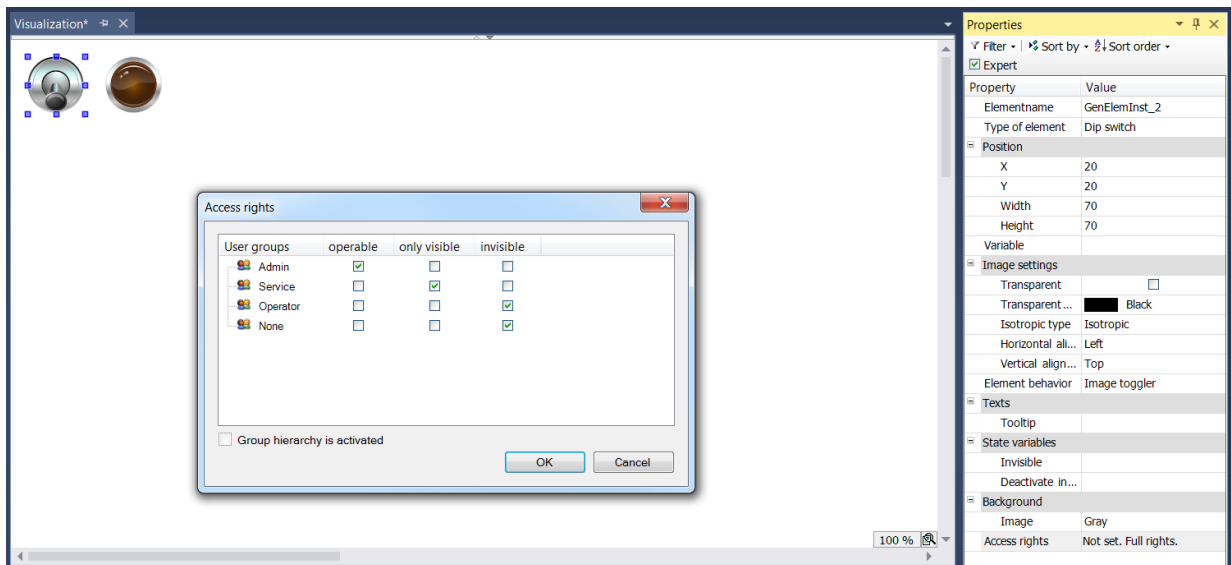
Programming the visualization

1. Configure your user management by defining [Groups](#) [▶ 396] and [Users](#) [▶ 397].

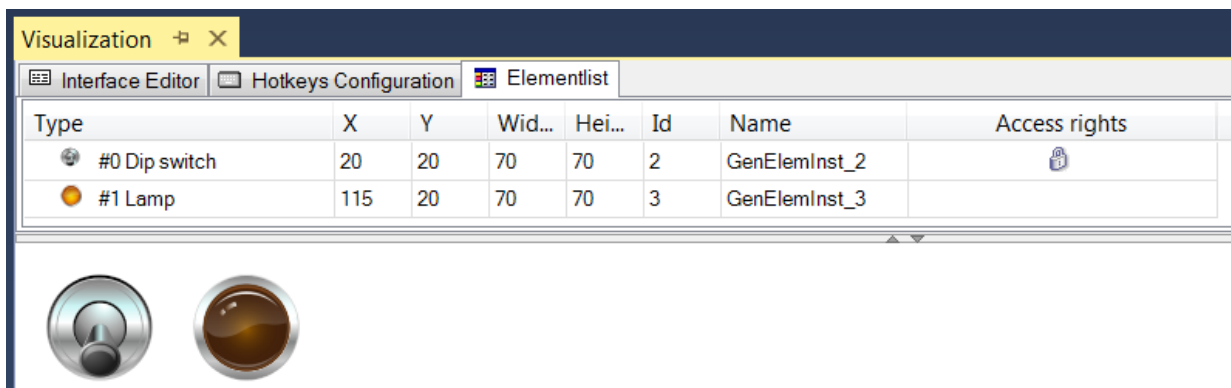
- Integrate the dialogs provided by the library "VisuUserManagement" in your visualization by programming elements with [Input configuration](#) [► 437]. A defined event triggers a dialog with a pre-programmed input configuration. You can program your own dialogs.



- Program the visualization elements by setting the property "Access rights" in the "Property" View. The behavior is then group-dependent.



The elements with limited rights are highlighted in the element list.



CSV file with user management data

The user management data are stored as a CSV file in the following format:

- User groups:

```
ID;group name;automatic logoff TRUE/FALSE;logoff time;unit logoff time;permission to change user data TRUE/FALSE
```

- User:

```
login name;full name;password encrypt TRUE/FALSE;password;group ID;user deactivated TRUE/FALSE
```

Use this format to edit the user management data with any tool of your choice. If "password encrypt" is set to FALSE, an unencrypted password can be entered, as shown for user "Hugo" in the example. After an import the password is immediately encrypted.

Sample:

V1.0.0.1

Usergroups:

1;Admin;TRUE;1;Minute;TRUE

2;Service;FALSE;5;Minute;FALSE

3;Operator;FALSE;1;Minute;FALSE

0;None;FALSE;1;Minute;FALSE

User:

HansM;Hans Mayer;TRUE;F9307D9940B6F7D78320E7E008377593;1;FALSE;administrator

PeterS;Peter Schmidt;TRUE;C5972629BF18E0E82D06FFF29B5BADFF;2|3;FALSE;team leader 1

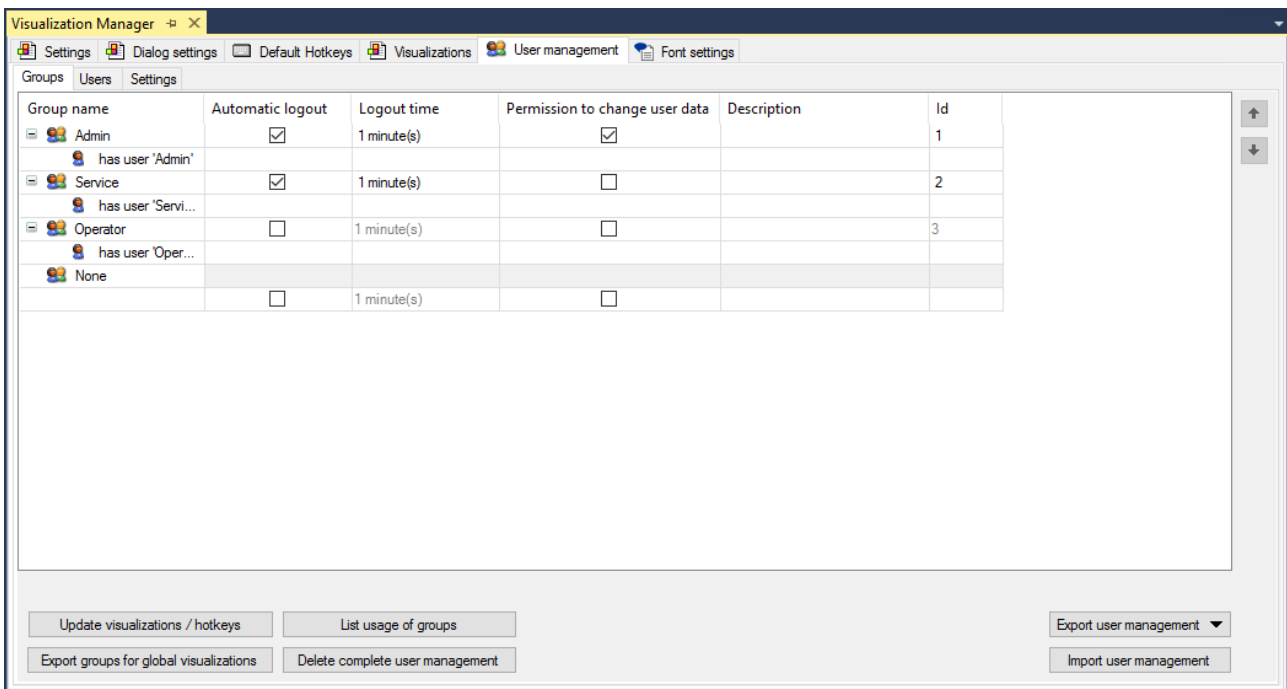
UllaM;Ulla Müller;TRUE;569D35AC3272623AECDDCA021916C2AB;2|3;FALSE;team leader 2

ElkeF;Elke Fischer;TRUE;C634F54AF9343142159FE0435D93929D;3;FALSE;operator team 1

PaulK;Koch;TRUE;01E2CBD4AE5442D9EACE33669549A3CC;3;FALSE;operator team 2



15.2.5.1 Groups

In "Groups" view, all groups are listed in hierarchical order.



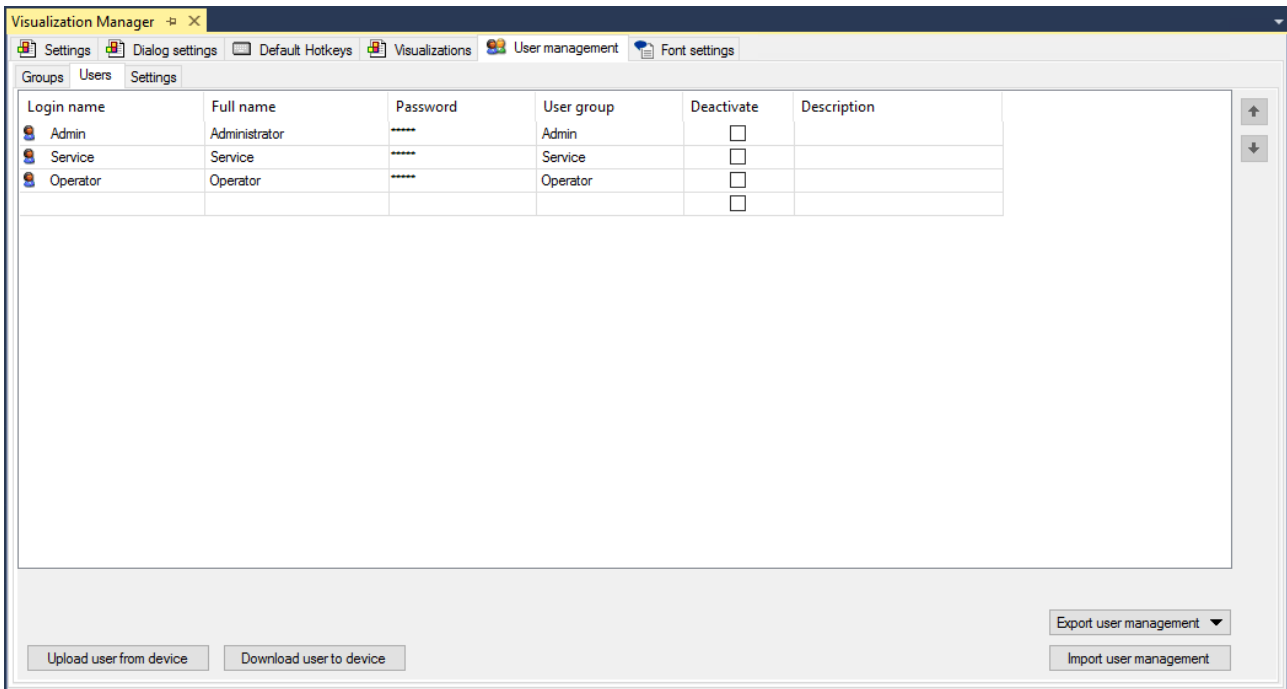
Group name	If the group node is expanded, all users that are allocated to this group are displayed.
Automatic logout	Tick the checkbox to define a fixed logout time.
Logout time	Enter the time after which the user is to be logged out. Use the line editor to enter a number, and use the selection list to set the unit.
Permission to change user data	Tick the checkbox to allow this group to edit user data when the visualization is online.
Description	Here you can enter comments or notes regarding the groups. This text is only available in the programming system and is not loaded in the runtime.
ID	Unique ID for each group. Issued automatically by the system.
Add group	To create a new group, click in the row at the end of the table in the Group column.
Delete group	Select a group by selecting a field in the corresponding table row. Press [Del] to delete the row and therefore the group associated with it. The group None cannot be deleted.

Buttons

Update visualizations / hotkeys	Opens the dialog "Update visualizations and hotkeys". Update if groups were changed at a point in time when visualizations or hotkeys already had restricted access possibilities.
List use of the groups	List of the visualizations and hotkeys with restricted access rights. The list is displayed in the Messages view.
Export groups for global visualizations	Click this button to transfer the groups defined above to Tools > Options > TwinCAT > PLC Environment > Visualization user management, where they are listed under "Use the following user group list".
Delete complete user management	The user management is deleted, although the access rights for the elements are persevered.
Export user management	A standard dialog opens, from which a CSV file can be saved in any directory with any name.
Import user management	A standard dialog opens, from which a CSV file can be loaded from any directory with any name
 , 	To organize the groups hierarchically in a certain order, click on the up arrow icon to move the group up one row or the down arrow icon to move the group down one row. The order in the table reflects the hierarchical order that can be loaded in this form into the runtime and is then available in online mode. A group from a higher hierarchy cannot have fewer access rights for an element than a group from a lower hierarchy.

15.2.5.2 Users

In "Users" view all user are listed.



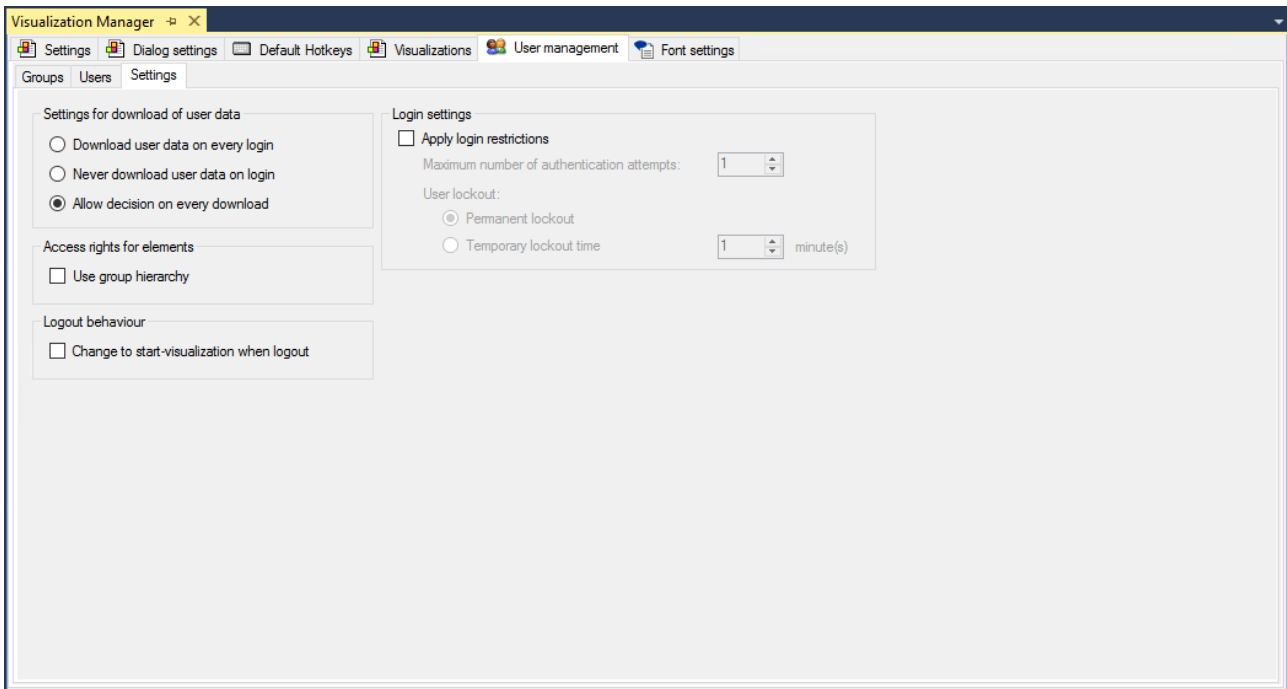
Login name	Enter a unique and compilable name with which the user can login on the visualization at runtime.
Full name	Enter the full name of the user. This may exist more than once in the user management.
Password	Enter a password, which will be encrypted. Each user is assigned their login name as their default password.
User group	Enter one or several group(s), to which the user belongs. Users who belong to several user groups can operate or view an element in the visualization when one of these groups has the relevant permission. Further details can be found in section Access rights [▶ 423].
Deactivate	Tick the checkbox to deactivate the user.
Description	You can enter comments and remarks about the user here. This text is only available in the programming system and is not loaded in the runtime.

Buttons

Load user from device	Upload user management data from the PLC. Any existing user data are overwritten.
Download user to device	Download user management data to the PLC. The existing user management is overwritten.
Export user management	A standard dialog opens, from which a CSV file can be saved in any directory with any name. The CSV file contains the data of the groups and user.
Import user management	A standard dialog opens, from which a CSV file can be loaded from any directory with any name and displayed here, provided the format is compatible.

15.2.5.3 Settings

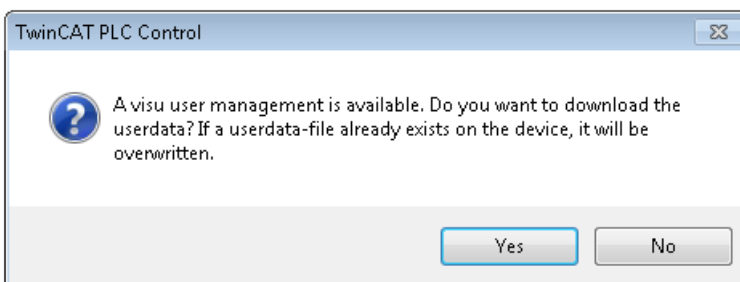
This screen shows the settings for downloading the user data.



Select a login procedure:

Settings for downloading the user data	
Download at each login	The user management data, which is stored in the programming system, is downloaded to the PLC at every login. Already existing data will be overwritten.
No download when logging in	The user management data is never downloaded, even if it has changed.
Decide again with each download	A dialog-controlled download is enabled.

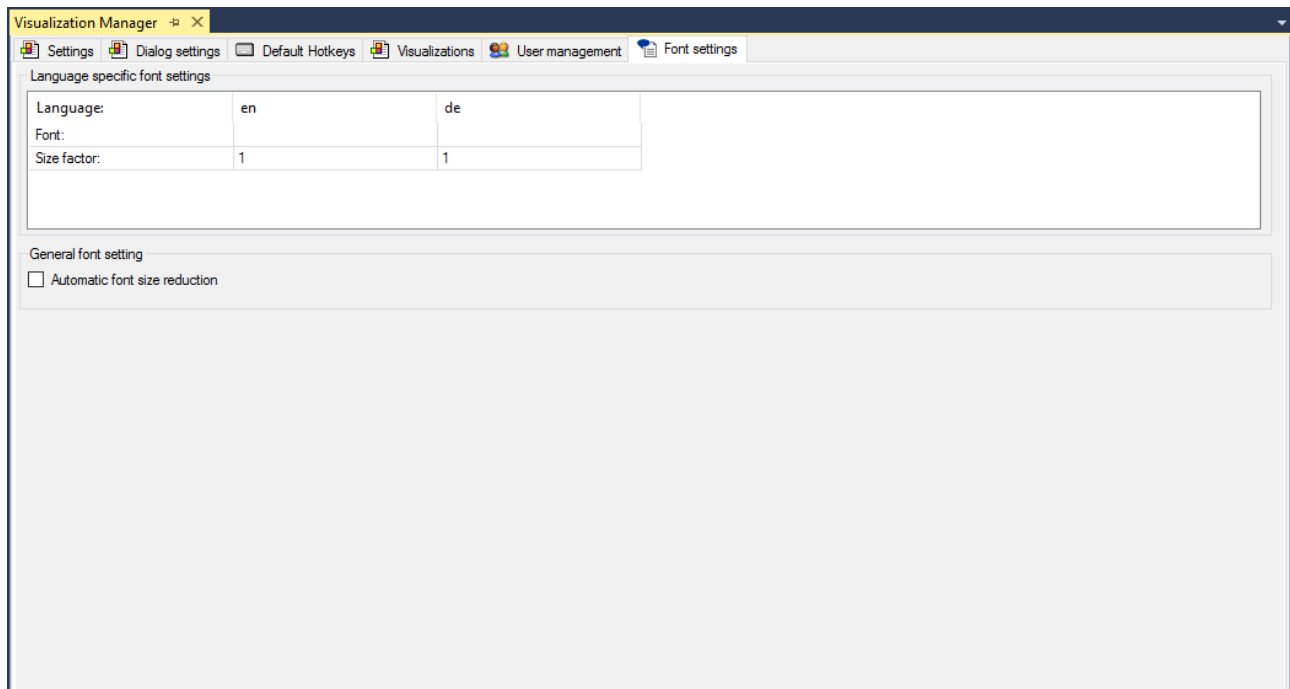
If the setting "Allow decision on every download" is selected, the following dialog is displayed before every download.



Access rights for elements	
Use group hierarchy	The groups are arranged hierarchically based on the Groups configuration. Accordingly, access rights can only be assigned hierarchically.
Logout behavior	
Switch to Start visualization on logout	Tick the checkbox if the start visualization is to open automatically when a user logs out.
Login settings	
Apply login restrictions	Tick the checkbox in order to use the settings "Maximum number of authentication attempts" and "User blocking" for the PLC visualization.
Maximum number of authentication attempts	Define the maximum possible number of authentication attempts. Possible values: [1 ... 10] If the maximum number of authentication attempts is reached, the user is temporarily or permanently blocked, depending on the "User blocking" setting.
User blocking	Select one of the two blocking modes: "permanent" or "temporary": <ul style="list-style-type: none"> • Permanent blocking: the user is permanently blocked. Users who have the permission to change user data cannot be permanently blocked. • Temporary blocking, time: specified in minutes, possible values [1 ... 2880]. Users who have the permission to change user data are always temporarily blocked for 10 minutes, even if a different value is entered here.

15.2.6 Font

This tab offers settings for adapting the font and font size of a text depending on the language. They apply to all visualizations in the project.



Language-specific font settings

Language	A list of preset languages is available. The languages used in the project extend this selection. All text lists of the project are scanned.
Font	The font used by the visualization depending on the language.
Size factor	The factor affects the font sizes of all texts in the visualization. If the factor is less than 1, the font size is made smaller. If the factor is 1, all texts are displayed unchanged, as defined in the properties. Preset: 1
Red highlighting of a cell	The setting cannot be used at runtime, since the corresponding language is no longer available in the text lists of the project or the libraries.

Context menu of a selected table row	
Delete	The corresponding column is removed. This is particularly advisable if settings in the column are highlighted in red.
Copy	All column settings are copied to the clipboard.
Paste	All column settings are overwritten with the values from the clipboard.

General font setting

Automatic reduction of the font size	If the text to be displayed does not fit inside the text box in the set format, the font size will be reduced automatically until the text fits completely inside the text box. Tip: This prevents a text not being fully displayed when switching to a language that needs more space. Provided a sufficiently small font is available.
--------------------------------------	---

See also:

- [Text and language in visualizations \[▶ 614\]](#)

15.3 Visualization Libraries

The [visualization elements \[▶ 404\]](#) programmed as function block are made available via libraries. If a [visualization object \[▶ 402\]](#) is added to a project, certain visualization libraries are integrated in the PLC project. The names and versions of these libraries are defined in the currently used [visualization profile \[▶ 402\]](#). The profile also precisely specifies which elements from these libraries are available in the [toolbox \[▶ 384\]](#) of the [visualization editor \[▶ 376\]](#).

A visualization library is always configured as a special type of a 'placeholder library'. As a result, the precise version of the library to be used is not specified, as long as it is not integrated in a project. Only then the current visualization profile determines which version is actually required. Note that this type of library differs from the device-specific placeholder libraries, in which the placeholders are resolved based on the device description.

See below for a description of the basic libraries, which are integrated by default, as soon as a visualization object is added to a standard project. They reference further libraries, which are not mentioned here. By default you do now have to explicitly add the visualization libraries or use them in your applications:

- System_VisuElem3DPath (only for TwinCAT 3.1 build <4020.0)
- System_VisuElemMeter
- System_VisuElems
- System_VisuElemsDateTime (only for TwinCAT 3.1 Build <4020.0)
- System_VisuElemsSpecialControls
- System_VisuElemsWinControls
- System_VisuElemTextEditor
- System_visuinputs
- System_VisuNativeControl

15.4 Preconditions

Since the TwinCAT 3 visualization is implemented based on the IEC61131-3 standard, certain [libraries](#) [▶ 401] have to be integrated in the project, which provide the required functions and visualization elements. [Visualization profiles](#) [▶ 402] are used to define the selection of the visualization libraries, and from these the selection of elements to be made available in the visualization editor object. When the first visualization page is added, the libraries specified by the profile are automatically added to the PLC project.

15.5 PLC project properties

The default settings can be modified in the PLC project properties. In the category "Visualization", for example, the default directories for text list files and image files can be modified. The dialog can be opened by right-clicking on the PLC project in the context menu under "Properties".

15.6 Visualization Profiles

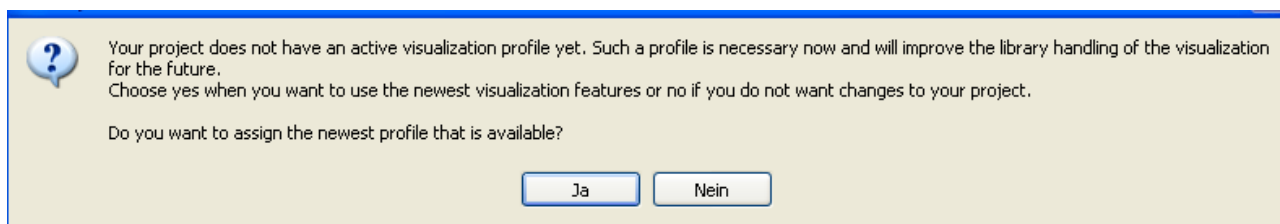
Each visualization project, i.e. a project containing at least one [visualization object](#) [▶ 402], must be based on a visualization profile. This profile defines the following:

- The names and versions of the [visualization libraries](#) [▶ 401], which are automatically integrated in the project when a visualization object is [created](#) [▶ 402].
- A selection of visualization elements originating from the integrated libraries, which are to be available in the [toolbox](#) [▶ 384] of the visualization editor.

The default profile for the project is defined in the [project properties](#) [▶ 402] under "Visualization Profile". It is possible to select a different profile at any time. Please note that the selected profile applies to all visualization objects of the project.

If a different profile is selected, a message appears to indicate that the profile may prevent logging in without online change or download. Changing the profile triggers an automatic update in the Library Manager with regard to the required libraries. In other words, the libraries do not have to be adapted manually.

If an old project is opened, which was not created based on a visualization profile, the system asks whether you wish to switch to the new profile mechanism, or not.



If "Yes", the latest profile is used. If "No", the oldest available profile is entered in the project settings (referred to as "compatibility profile"), but no further changes are made in the project.

15.7 Visualization object

A [visualization](#) [▶ 376] in a PLC project can comprise different visualization pages. A visualization page can be [used](#) [▶ 612] in another one, and it is possible to [switch](#) [▶ 613] between different pages at runtime. Each of these visualization pages is represented by a separate visualization object.

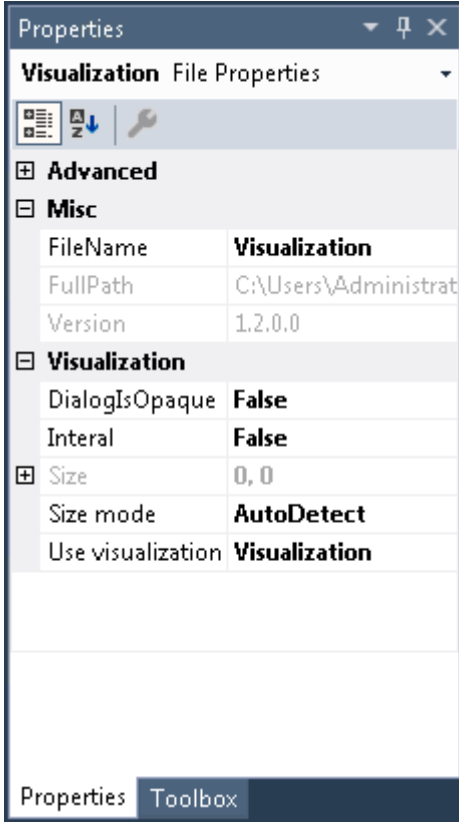
Creating a visualization object

To create a visualization object in a PLC project, right-click on the PLC project or a folder within it (e.g. "Visus") to open a context menu in the PLC project. Then use the menu item "Add" to select a visualization object.

When the first visualization object is added, the [visualization libraries \[▶ 401\]](#) and the [Visualization Manager \[▶ 387\]](#) are automatically added to the project. The newly created visualization object is opened in the [visualization editor \[▶ 376\]](#).

Settings for a visualization object

If a visualization object is selected, its settings are displayed in the Properties window as follows:



Misc

FileName	Name of the visualization object
FullPath	Memory location of the object. The memory location cannot be changed at this point. It is therefore grayed out.

Visualization

Dialog Is Opaque	<p>If this setting is active the screen area hidden by this dialog is not refreshed. This has a positive effect on the drawing and input performance.</p> <p>Note: Use this option only if the dialog you have drawn is rectangular and fully opaque, i.e. it contains no transparent parts.</p>
Internal	<p>A visualization marked as internal is visible and usable as usual inside a PLC project. If this visualization is saved in a library it is no longer visible or usable in the PLC project in which the library is used.</p>
Size	<p>Size of the visualization page –the setting is grayed out if the entry "AutoDetect" has been selected under "Size mode".</p> <ul style="list-style-type: none"> • Width: width in pixels • Height: height in pixels
Size mode	<p>Here you can specify whether and how the size is to adapt to the visualization page.</p> <ul style="list-style-type: none"> • AutoDetect: The size of the visualization page is determined, at which all currently included visualization elements are visible. • AutoDetectWithBgImage: The size of the visualization page is determined, at which all currently included visualization elements and the background image [▶ 378] are visible. • Specified: The size of the page is specified under "Size". The user should consider whether all visualization elements - and the background image, where applicable - fit into this section and are therefore fully visible. <p>If the aspect ratio resulting from these settings does not match the screen size, the visualization page is displayed with corresponding white edges. In this way distortion of the visualization elements is prevented.</p>
Use visualization as	<p>In this setting one of the following visualization types can be specified for the object:</p> <ul style="list-style-type: none"> • Visualization: With this default setting, the visualization object is declared as a separate visualization page. • Dialog: The visualization object represents a dialog. If this setting is enabled, the visualization object can be used as a dialog [▶ 409] in other visualization objects. • NumKeypad: The visualization object represents a numpad/ a keypad. If this setting is enabled, the visualization object can be used as a numpad or keypad, e.g. for describing a variable [▶ 417] in the visualization. The interface of such a numpad/ keypad must look exactly like that of the default numpad/default keypad provided by the library "VisuDialogs".

Editing a visualization object

The [visualization editor \[▶ 376\]](#) for creating visualizations works in combination with the [toolbox \[▶ 384\]](#), which provides the visualization elements from the integrated element libraries, and the [properties editor \[▶ 385\]](#) for configuring the visualization elements. To open a visualization object, double-click on the object in the project tree.

Defining a start visualization

The "start visualization", i.e. the visualization object that is to be displayed first at the start of a [PLC HMI \[▶ 603\]](#) or [PLC HMI Web \[▶ 608\]](#) client, must be entered in the [TargetVisualization \[▶ 606\]](#) or [WebVisualization \[▶ 609\]](#) object.

15.8 Visualization elements

The different visualization elements are available in the [toolbox \[▶ 384\]](#), once a [visualization object \[▶ 402\]](#) has been opened. They are divided into the following categories:

- [Common Controls \[▶ 421\]](#)
- [Basic \[▶ 475\]](#)
- [Lamps/Switches/Images \[▶ 527\]](#)

- [Measuring devices \[▶ 537\]](#)
- [Special controls \[▶ 580\]](#)

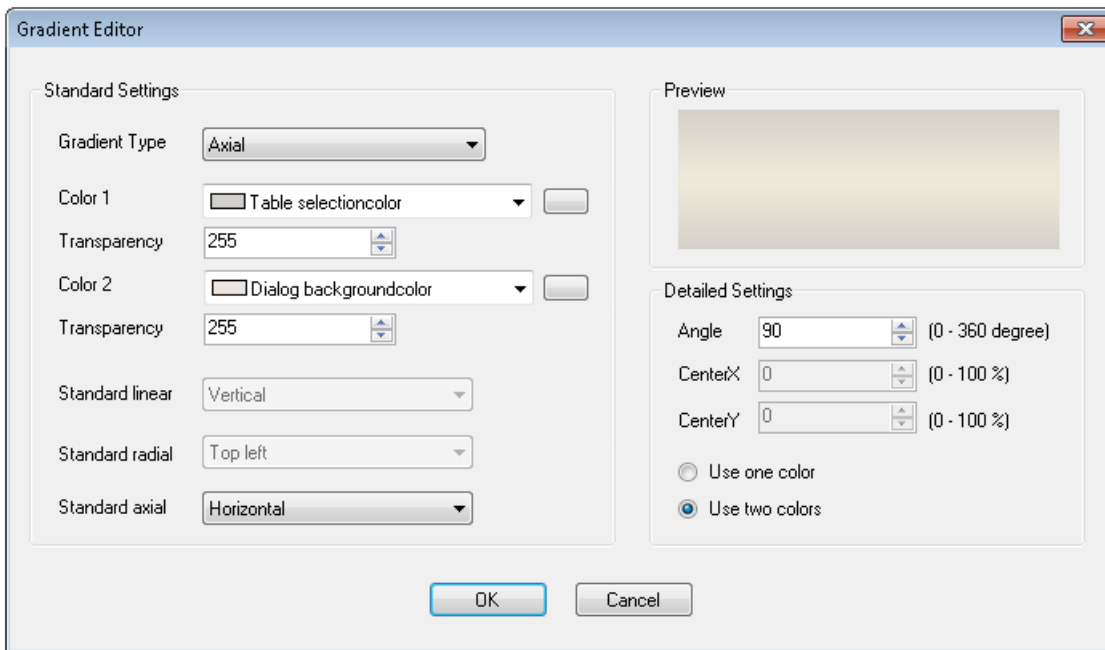
15.8.1 General configuration

The following three general dialogs are available, which are used for different visualization elements:



- [Color gradient dialog \[▶ 405\]](#)
- [Input configuration \[▶ 406\]](#)
- [Access rights dialog \[▶ 420\]](#)

15.8.1.1 Gradient editor

Click in the value field for the color gradient in the Properties window to open the gradient editor.



Default settings

Gradient Type	The three following gradient types are available for selection: <ul style="list-style-type: none"> • Linear • Radial • Axial
Color 1	First gradient color – This can be selected from the combo box or the color dialog, which can be opened via the button  .
Transparency	Values between 0 and 255 can be used to specify the "strength" of the first color.
Color 2	Second gradient color – This can be selected from the combo box or the color dialog, which can be opened via the button  .
Transparency	Values between 0 and 255 can be used to specify the "strength" of the second color.
Standard linear	Default settings for the linear color gradient type (gradient angle): <ul style="list-style-type: none"> • Horizontal • Vertical • From top left • From top right • Horizontal color switched • Vertical color switched • From bottom left • From bottom right
Standard radial	Predefined settings for the radial color gradient type (position of the center): <ul style="list-style-type: none"> • Center • Top left • Top right • Bottom left • Bottom right
Standard axial	Predefined settings for the axial color gradient type (gradient angle): <ul style="list-style-type: none"> • Horizontal • Vertical • From top right • From top left

Detailed settings

Angle (degrees)	Available only for gradient types linear and axial
Center X (%)	X-position of the center (0-100%) – available only for color gradient type radial.
Center Y (%)	Y-position of the center (0-100%) – available only for color gradient type radial.
Use one color	Color gradient type between color 1 and the same color with a different brightness. The brightness can be set between 0 (black) and 100 (white).
Use two colors	Type of color graduation between the two selected colors "Color 1" and "Color 2".

15.8.1.2 Input configuration

One or several of the subsequent actions described below can be defined in the input configuration dialog for a certain event [[▶ 437](#)] (e.g. OnMouseClicked). Click in the value field for this event in the Properties window to open the dialog. The name of the event, for which these subsequent actions are selected, is shown as title in the grey row.

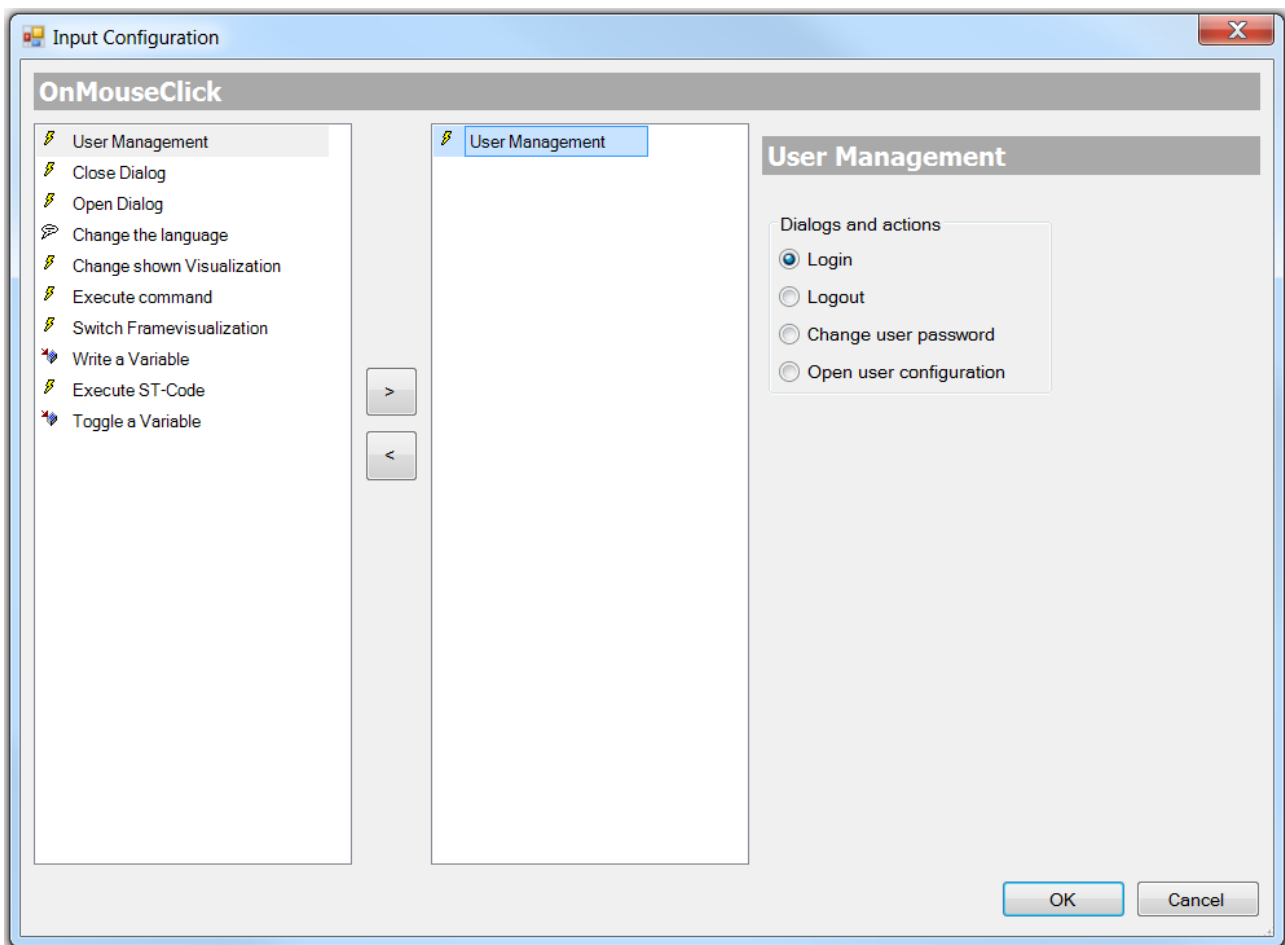
To add a subsequent action, the corresponding action must first be selected on the left-hand side of the dialog. The action can then be added by pressing the button with the right arrow icon. To delete an action, select it on the right-hand side and press the button with the left arrow icon.

The following actions are available:

- [User management](#) [▶ 407]
- [Close dialog](#) [▶ 409]
- [Open dialog](#) [▶ 409]
- [Language switching](#) [▶ 411]
- [Change shown Visualization](#) [▶ 411]
- [Execute command](#) [▶ 412]
- [Switch frame visualization](#) [▶ 414]
- [Write a variable](#) [▶ 417]
- [Execute ST-Code](#) [▶ 419]
- [Toggle a Variable](#) [▶ 420]

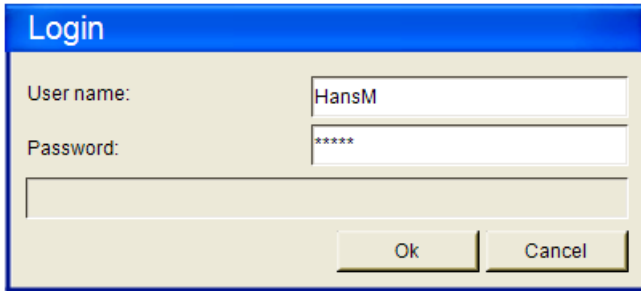
User management

User management can only be selected as subsequent action, if a [User management](#) [▶ 393] was created first or the library "VisuUserManagement" was added manually.



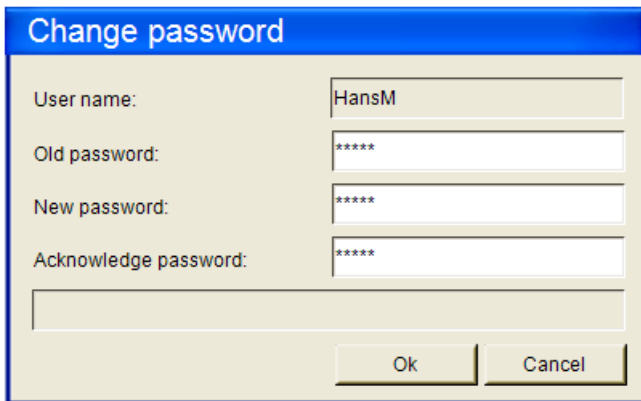
Via the user management the following four standard dialogs and actions can be added. It enables the visualization user in online mode to:

- login



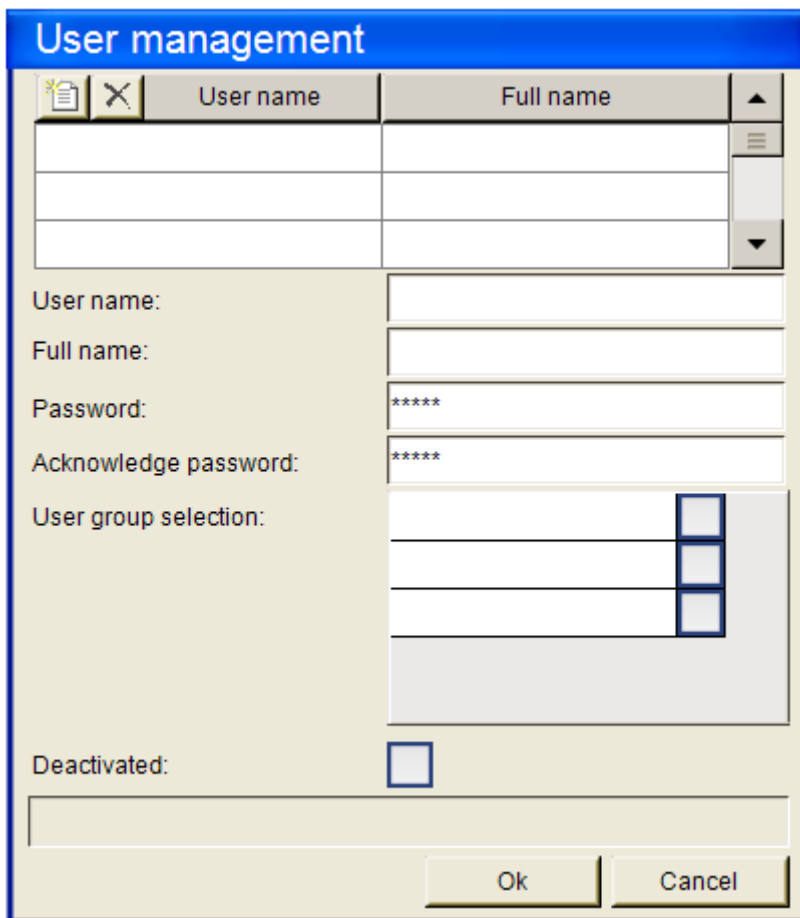
The 'Login' dialog box features a blue title bar. It contains two input fields: 'User name:' with the text 'HansM' and 'Password:' with six asterisks. Below these fields is a larger empty text area. At the bottom right, there are 'Ok' and 'Cancel' buttons.

- logout
- change user password



The 'Change password' dialog box has a blue title bar. It includes four input fields: 'User name:' (HansM), 'Old password:' (*****), 'New password:' (*****), and 'Acknowledge password:' (*****). A larger empty text area is located below the fields. 'Ok' and 'Cancel' buttons are positioned at the bottom right.

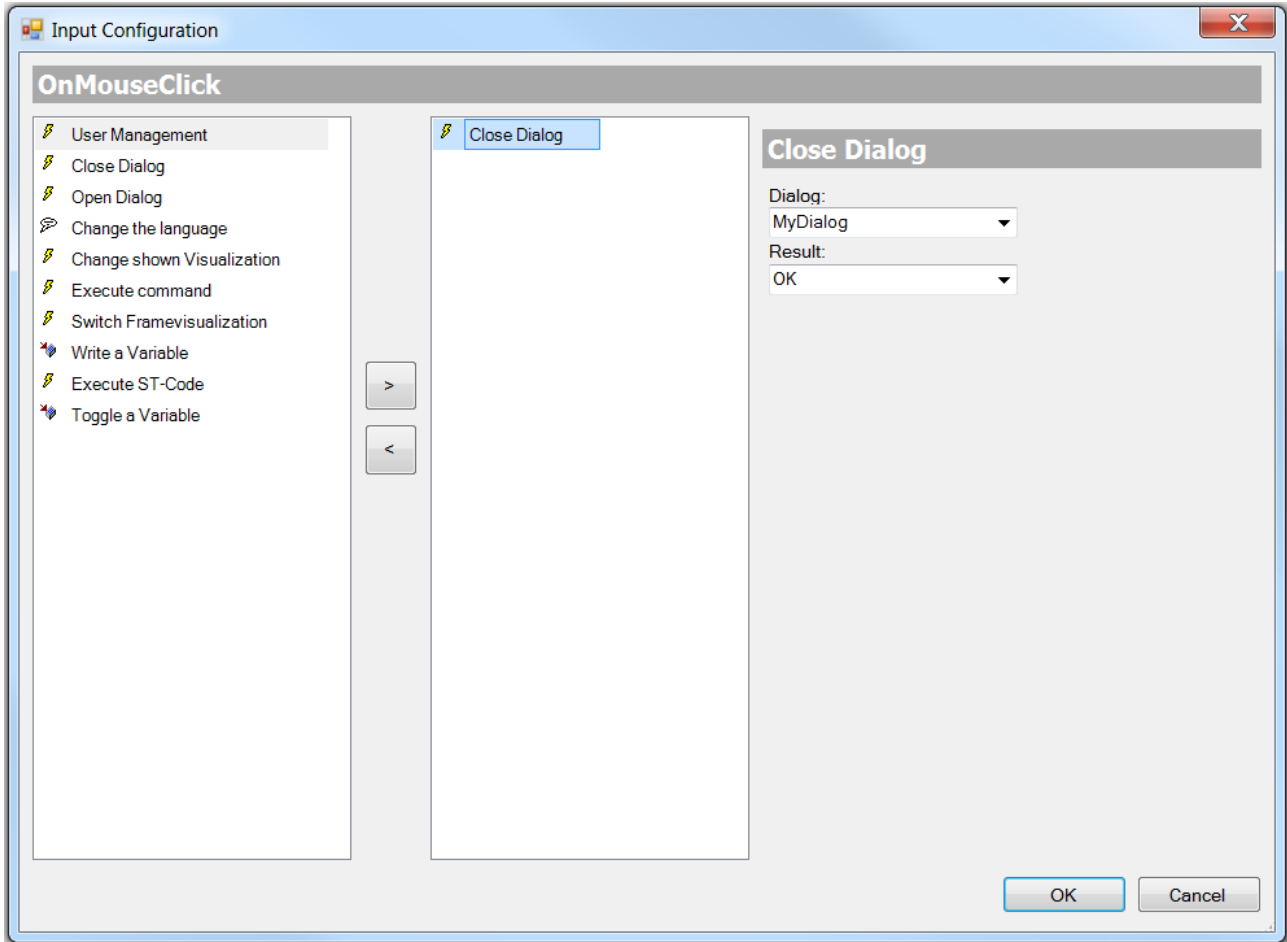
- open user management



The 'User management' dialog box has a blue title bar and a toolbar with a document icon and a close icon. It features a table with two columns: 'User name' and 'Full name'. The table is currently empty. Below the table are several input fields: 'User name:', 'Full name:', 'Password:' (*****), and 'Acknowledge password:' (*****). A 'User group selection' section contains a list box with three empty entries. At the bottom, there is a 'Deactivated:' checkbox which is unchecked, followed by a larger empty text area. 'Ok' and 'Cancel' buttons are at the bottom right.

Close dialog

This option can be used to specify that after the event the specified dialog should be closed with the specified result. Select the required dialog from the selection list, which offers all currently available input dialogs.



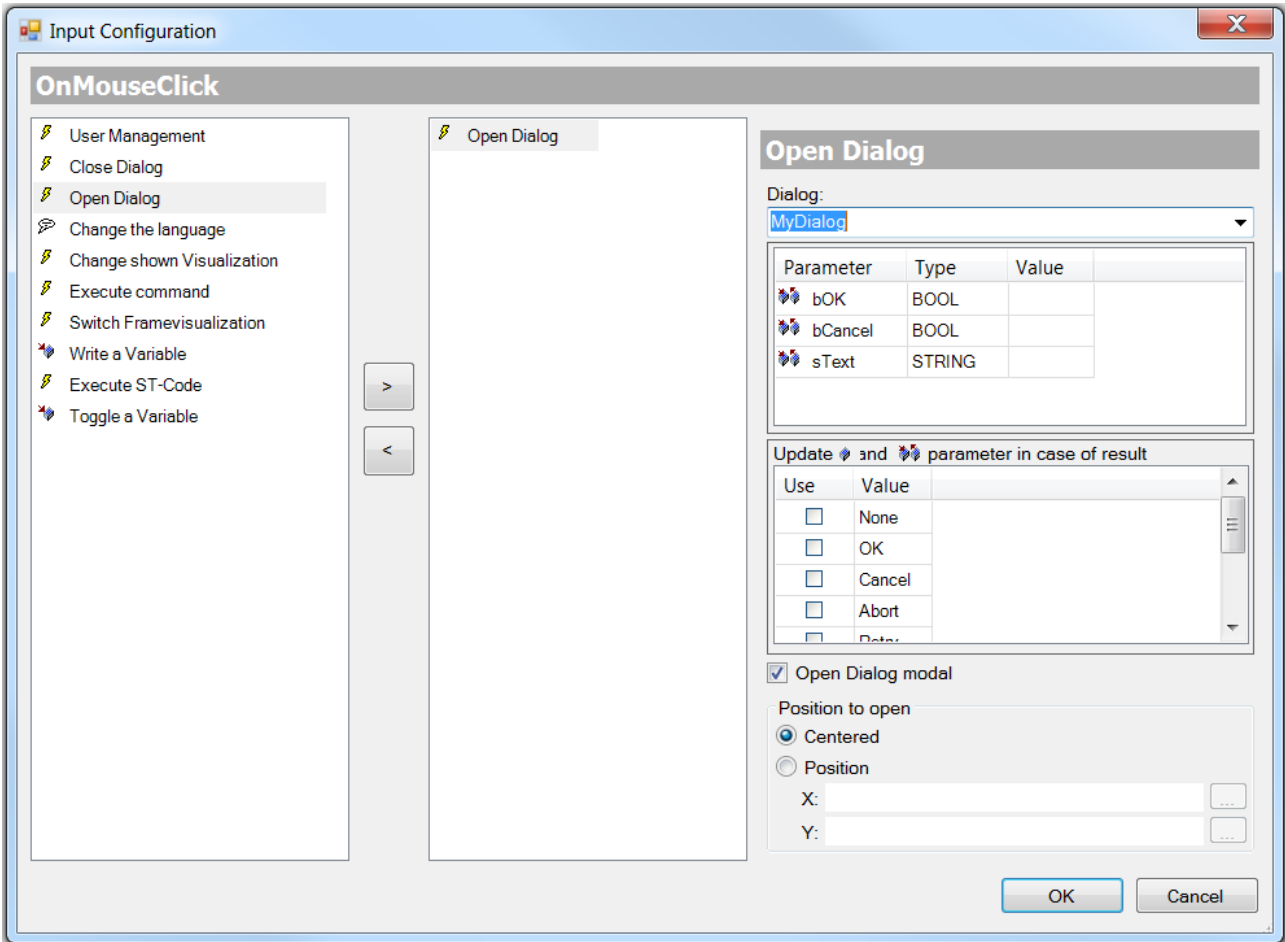
Dialog	Select a dialog from the selection list
Result	The list offers the standard options used in dialogs. These standard options require a user response. <ul style="list-style-type: none"> • OK • Cancel • Abort • Retry • Ignore • Yes • No

i Not that the result from the dialog that was closed last can be retrieved, and that a corresponding response can then be configured in any element of the same visualization. Use the configuration option "OnDialogClosed" for this purpose.

Open dialog

Here you can define that a mouse action is to be followed by opening of a dialog, which is represented by another (standard or user-created) visualization. The selection list offers all visualizations, for which the purpose "Dialog" is entered in the object properties [▶ 403]. It enables a user-created dialog to be used as user input mask in a visualization.

As a minimum, the standard objects "VisuDialogs.Login" and "VisuDialogs.FileOpenSave" are available, if VisuDialogs.library is integrated in the project. In addition, user-created input dialogs can be used.




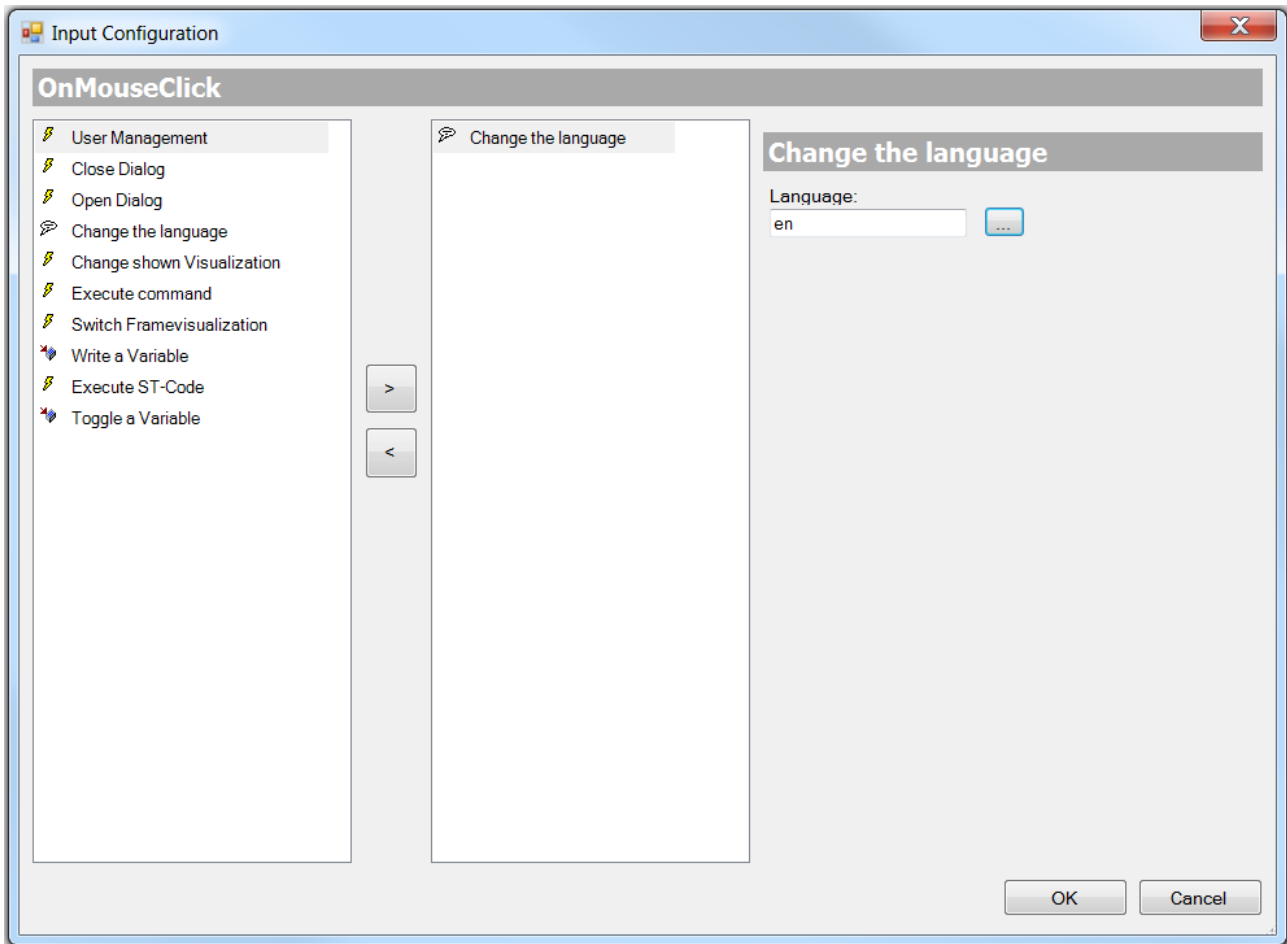
Dialog	Select a dialog from the selection list
Parameter	For the selected dialog visualization the input (VAR_INPUT), output (VAR_OUTPUT) and input/output parameters (VAR_IN_OUT,), which were defined in the interface editor [▶ 378] , are shown. (Parameter, Type, Value) In the last column, i.e. "Value", the parameters can be linked with variables from the PLC. The variable values are written to the parameters (VAR_INPUT, VAR_IN_OUT) when the dialog visualization is opened. (VAR_OUTPUT, VAR_IN_OUT).
Event	The user has to specify which dialog visualization result is to be used as a basis for writing its output and input/output parameters: tick the Use column to activate the required result value. Possible values: <ul style="list-style-type: none"> • None • OK • Cancel • Abort • Retry • Ignore • Yes • No



Note that the output and input/output parameters of a dialog visualization are not written until the dialog is closed! Until then, the values are only stored on the stack, i.e. they are not treated as references but as copies.

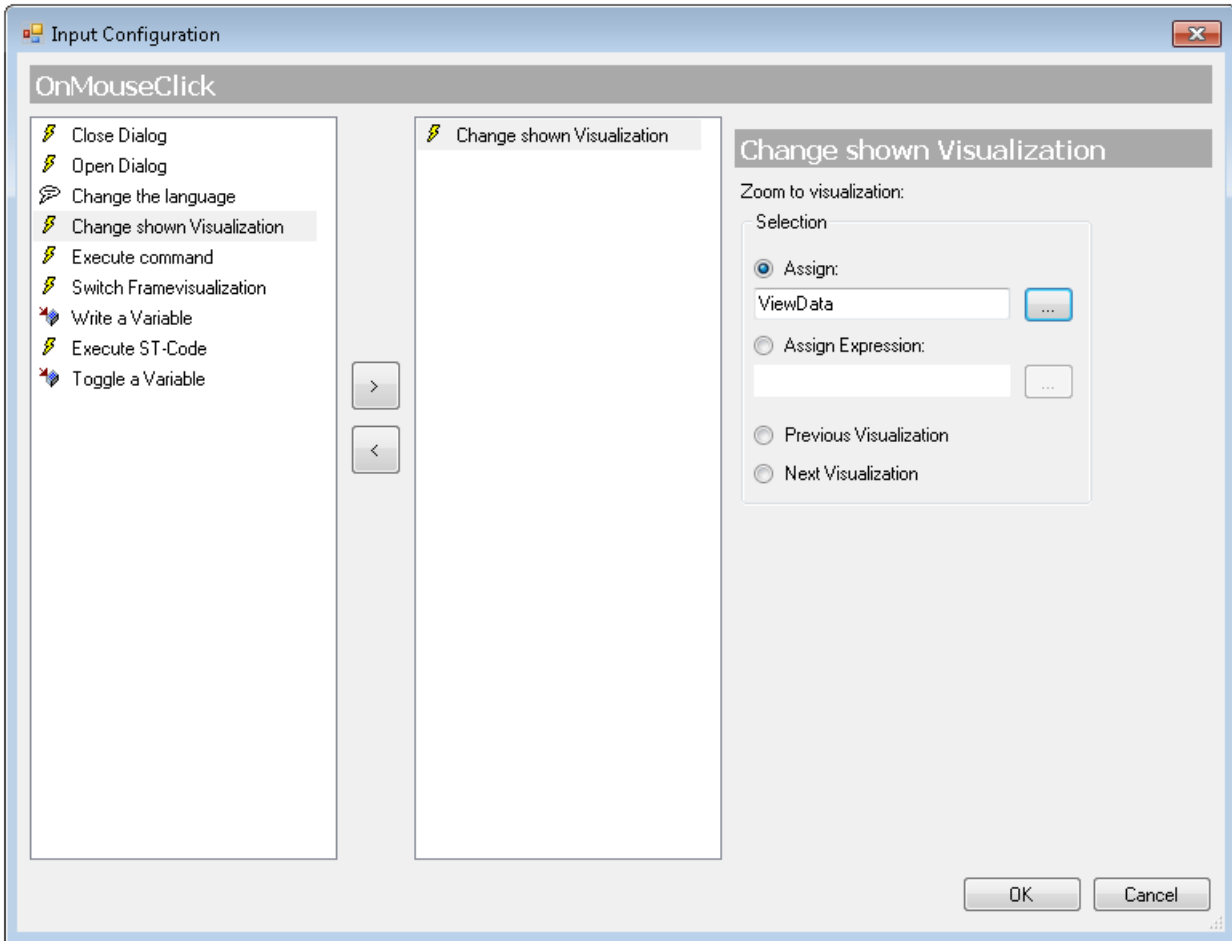
Language switching

Here you can specify an interface language for displaying the visualization texts. Use the language code, that is used in the corresponding [text list\(s\)](#) [[▶ 138](#)]. Alternative, click on the  button to select the required language in the input wizard. (See also [Language and Text](#) [[▶ 614](#)])




Change shown Visualization

Here you can select a change in the currently displayed visualization page as subsequent action. (Refer also to the section "Switching between visualization pages")

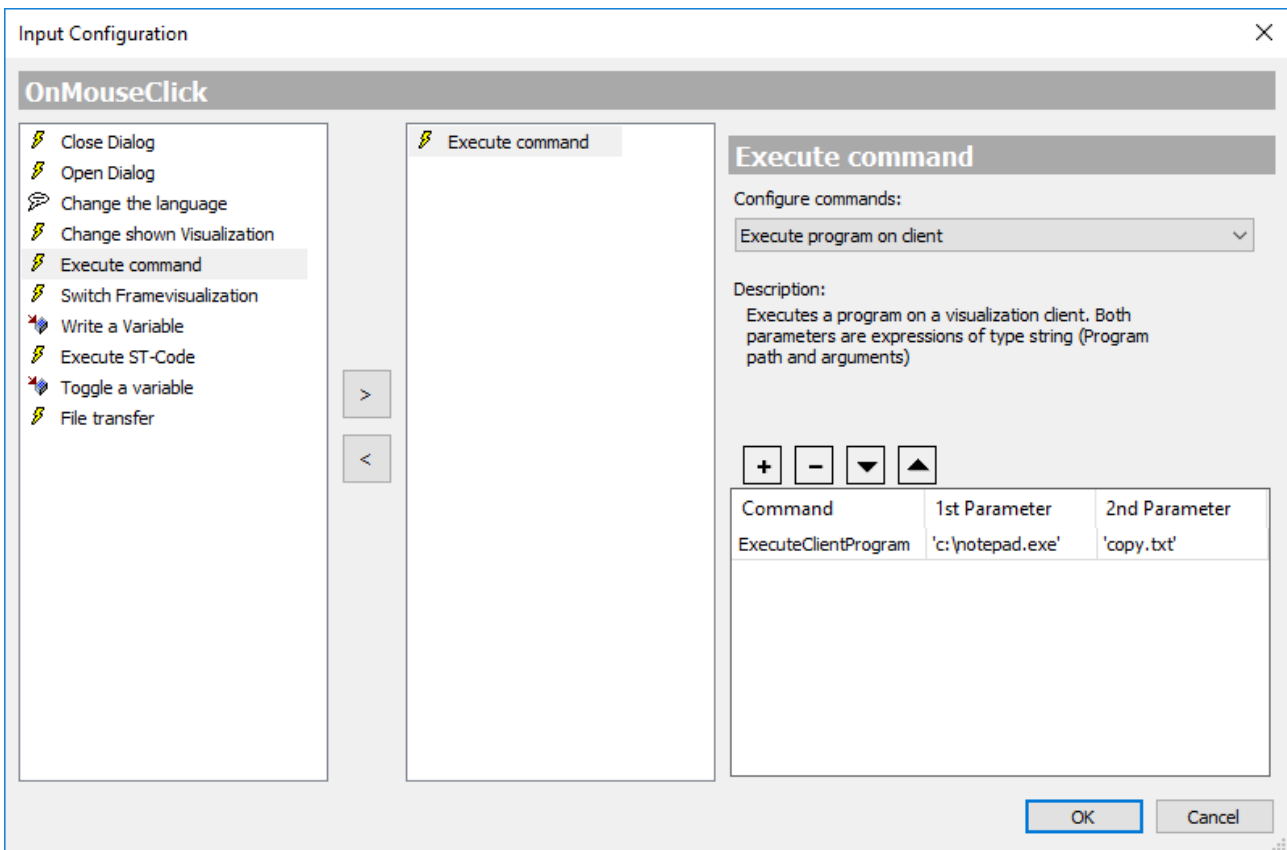



Select one of the following options to define which visualization page is to be displayed upon the mouse action in online mode:

Assign	The visualization page can entered directly. It is best to use input wizard, which can be opened via the button  .
Assign Expression	Here you can specify a variable of type String, which is used by the PLC project and provides the name of the visualization page. Example: <code>sVisualizationName : STRING := 'MyVisualization';</code>
	The order in which the visualization pages are displayed sequentially via user inputs is stored internally. This information can be used with the following two options.
Previous Visualization	The previously displayed visualization page is displayed again. If no visualization was called before, the current one continues to be shown.
Next Visualization	The visualization page listed next in the recorded sequence of visualization changes is displayed. This is therefore only possible, if a previous visualization change has already taken place via "Previous Visualization".


Execute command

Here you can define one or several commands to be executed following the mouse action.



Select a command from the "Configure commands" list and press the  button to added to the table in the lower section of the dialog. This table contains all selected commands of the input configuration.

A brief description of the command selected in the list is shown in the middle section of the dialog. Add the command parameters in the table columns "1st Parameter" and "2nd Parameter". The "Command" column shows the internal command name. See also the following table for a description of the individual commands.

Use  button to remove the currently selected entry from the list. Subsequently, when the user input for the visualization element takes place, the configure commands are executed one after the other from top to bottom according to their arrangement in the table. To change the order, the entries can be moved with the

 and  buttons.

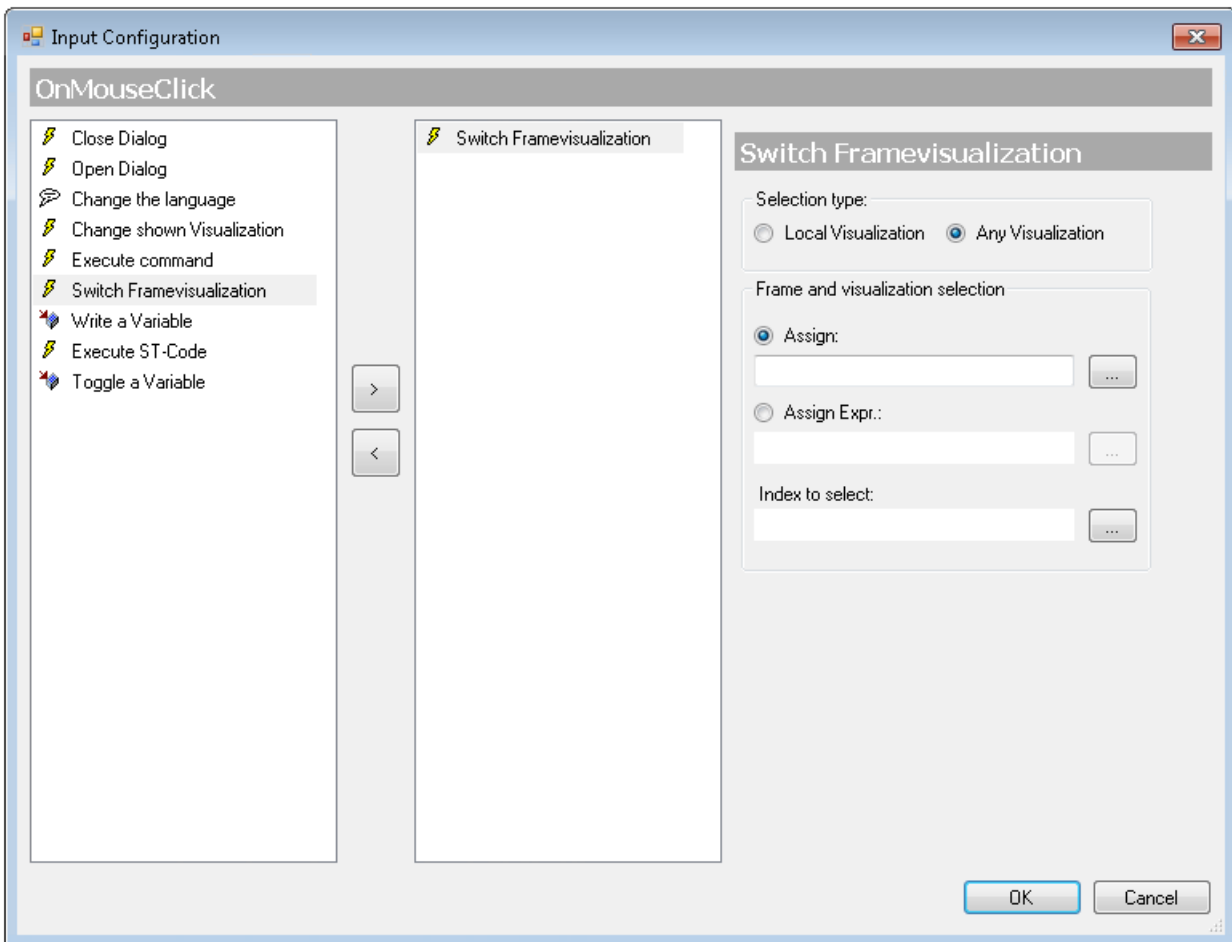
The following commands are available:

Execute program on controller Execute program on client	The specified program (*.exe) is executed on the controller or the visualization client. 1st parameter: string, path of the program file (example: "c:\programs\notepad.exe") 2nd parameter: string, arguments for the program to be executed, e.g. name of a file to be opened by the program (example: copyfile.txt)
Print	The standard dialog "Print" is opened, where settings can be made for the setting of the printer area and the printer parameters, etc. Moreover, the current visualization can be printed out. This command is only supported within the PLC HMI under Windows.
Navigate to URL (WebVisu)	Requirement: visualization is executed as PLC HMI Web. When the input event occurs, the visualization navigates to the web page with the specified URL 1st parameter: Web address URL - as variable of type string, to set the start page programmatically. (Example: MAIN.sUrl) - as literal in single quotation marks. (Example: http://www.beckhoff.com) 2nd parameter: If no parameter is specified, the web page is displayed in a new window or a new tab. If "replace" is specified, the PLC HMI Web is replaced with the web page.

Switch frame visualization

Requirement: A visualization in a project has at least one frame element, which was assigned various visualizations via the [Frame selection \[▶ 525\]](#). Within the frame the visualizations are indexed 0, 1, 2, etc. By default, the first of the allocated visualizations (Index 0) is displayed online in the frame.

In the present dialog you can now link a mouse action on the current visualization element in a certain frame element with displaying a certain associated visualization. The current element can therefore be used for programming a visualization switch-over in a frame. (Refer also to the section "[Switching between visualization pages within a frame \[▶ 612\]](#)")



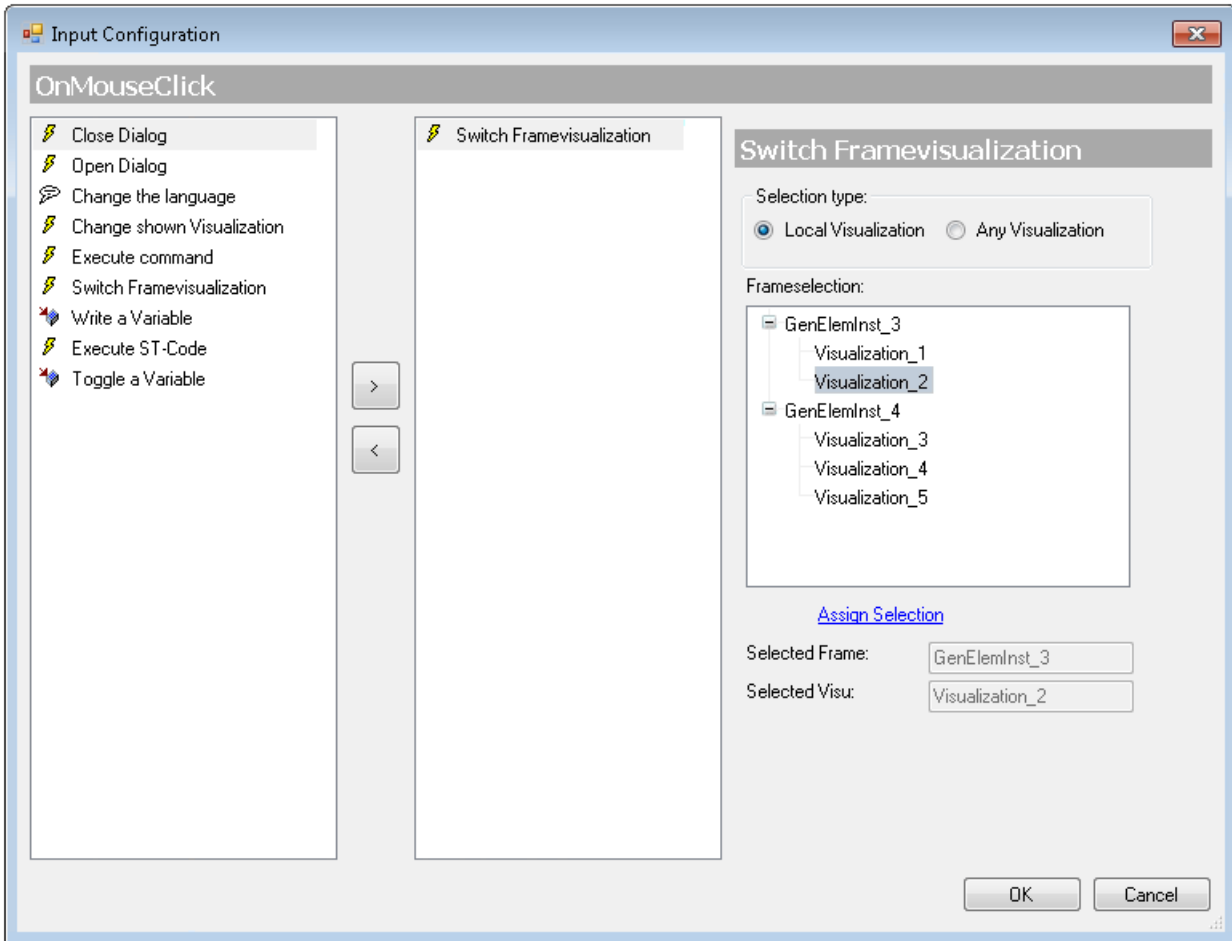
Selection type of the affected frame element

- Can be limited to the current visualization, resulting in a simpler configuration dialog, although without dynamic configuration option. (→ Local Visualization)
- Can be extended to all visualizations available in the project, including dynamic definitions of the frame and the displayed visualization (→ Any Visualization)

Local Visualization

Only frames of the current visualization can be addressed and are therefore available here in the field "Frame selection" (see below). This is a simple dialog for fast, direct configuration in the local visualization. However, it is not possible to specify the required visualization via a project variable. If this is required, select "Any Visualization".

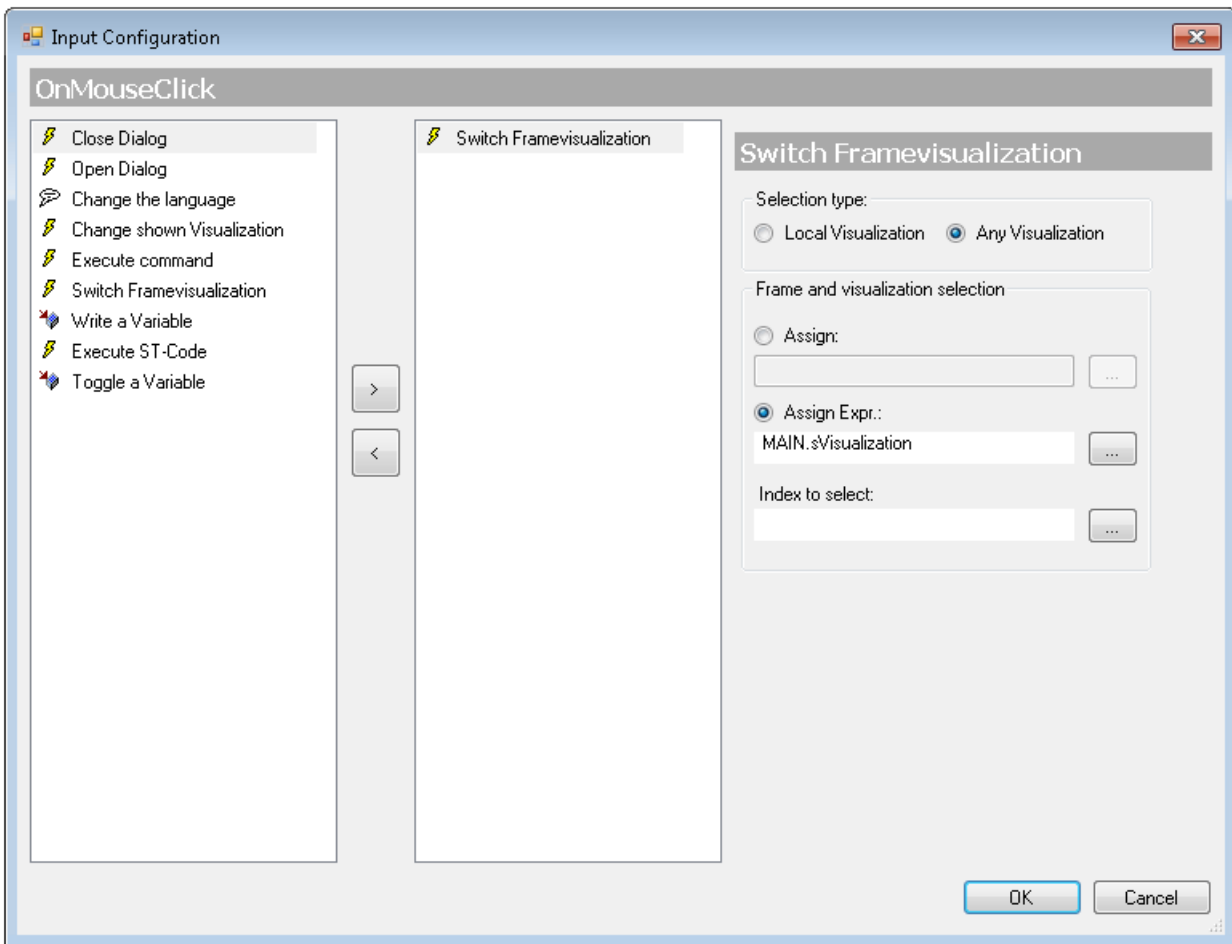
The frame selection shows the local frame elements and indented below the associated visualizations.



Select the required visualization from the affected frames, which is to be displayed following the mouse action. Click on "Assign Selection" to save the settings. The current selection is then displayed in the fields Selected Frame and Selected Visu.

Any Visualization

All frames of the project and their allocated visualizations are available. In this case frame and visualization can also be specified via project variables, i.e. dynamically.

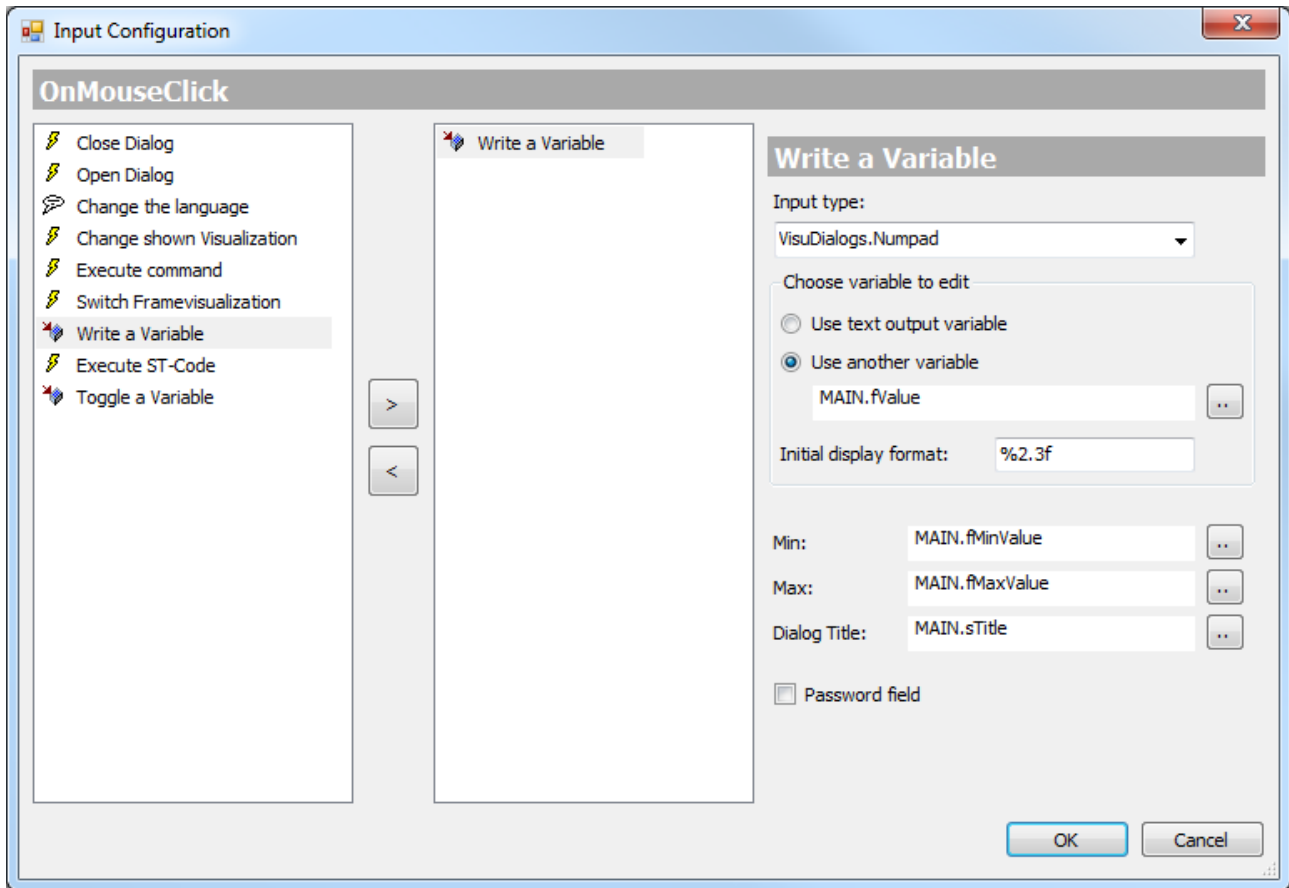


The "Frame and visualization selection" is then made via the following fields and options:

Direct assignment	If this option is enabled, the name of the affected frame element can be entered directly with the corresponding visualization. Press <input type="text" value="..."/> to invoke the input assistant. Example: Visualization_1.MyFrame
Assignment via expression	This option can be enabled in order to use a project variable of type string for specifying a frame element. Press <input type="text" value="..."/> to invoke the input assistant. The variable must then return the name of the frame element and the corresponding visualization name. Example: sVisualization : STRING := 'Visualization_1.MyFrame';
	The visualization to be displayed following the mouse action in the selected frame is specified via its index. This visualization index is an ascending integer number starting with 0, which was allocated to a frame in the frame selection [525] . By default, the first visualization is therefore always displayed in this list initially.
Index to select	Enter the index of the required visualization directly or via a project variable in the application. Press <input type="text" value="..."/> to open the input assistant for selecting a variable.

Write a variable

If an input configuration of type "Write a variable" exists for a visualization element, the element will offer an option to enter a value, as soon as the corresponding mouse action is executed. The value can then be entered as a string via a numpad or keypad and is then written to the project variable that is specified here in the input configuration dialog. The value is interpreted as text or a number, according to the data type of the project variable.




Select one of the following input types from the selection list on the right of the dialog:

Standard	The default input type is used. The default can be specified in the settings of the Visualization Manager.
Text input	A field for the value/text input opens. A keyboard is required to enter values.
Text input with limits	A field for the value/text input opens. In addition, the input limits are displayed at the top. They are shown in red font if they are exceeded. A keyboard is required to enter values.
VisuDialogs.Keypad	The mouse action opens a simulated keyboard. You can enter a string by clicking on the corresponding keys.
VisuDialogs.Numpad	The mouse action opens a simulated numpad. You can enter a number sequence by clicking on the corresponding keys.
	In addition to the default input types, all visualizations are offered, which were defined in their properties as "numpad", "keyboard" or "dialog for input configuration". (see Object properties [▶ 403])



In order to be able to use the keypad or numpad, the library "VisuDialogs" must be added in the library manager.

Select the variable to be edited:

Use text output variable	The value is written to the variable that is specified as text output variable in the visualization element.
Use another variable	Specify a project variable. Press  to open the input assistant.
Initial display format	Specification of a placeholder with formatting data Example: %2.3f
Min	Lower limit for the variable to be entered
Max	Upper limit for the variable to be entered
Dialog Title	Dialog title to be displayed in the dialog header, can be specified as text or via a text variable
Password field	To show "*****" instead of the text (e.g. in the case of passwords), tick the checkbox at Password field.

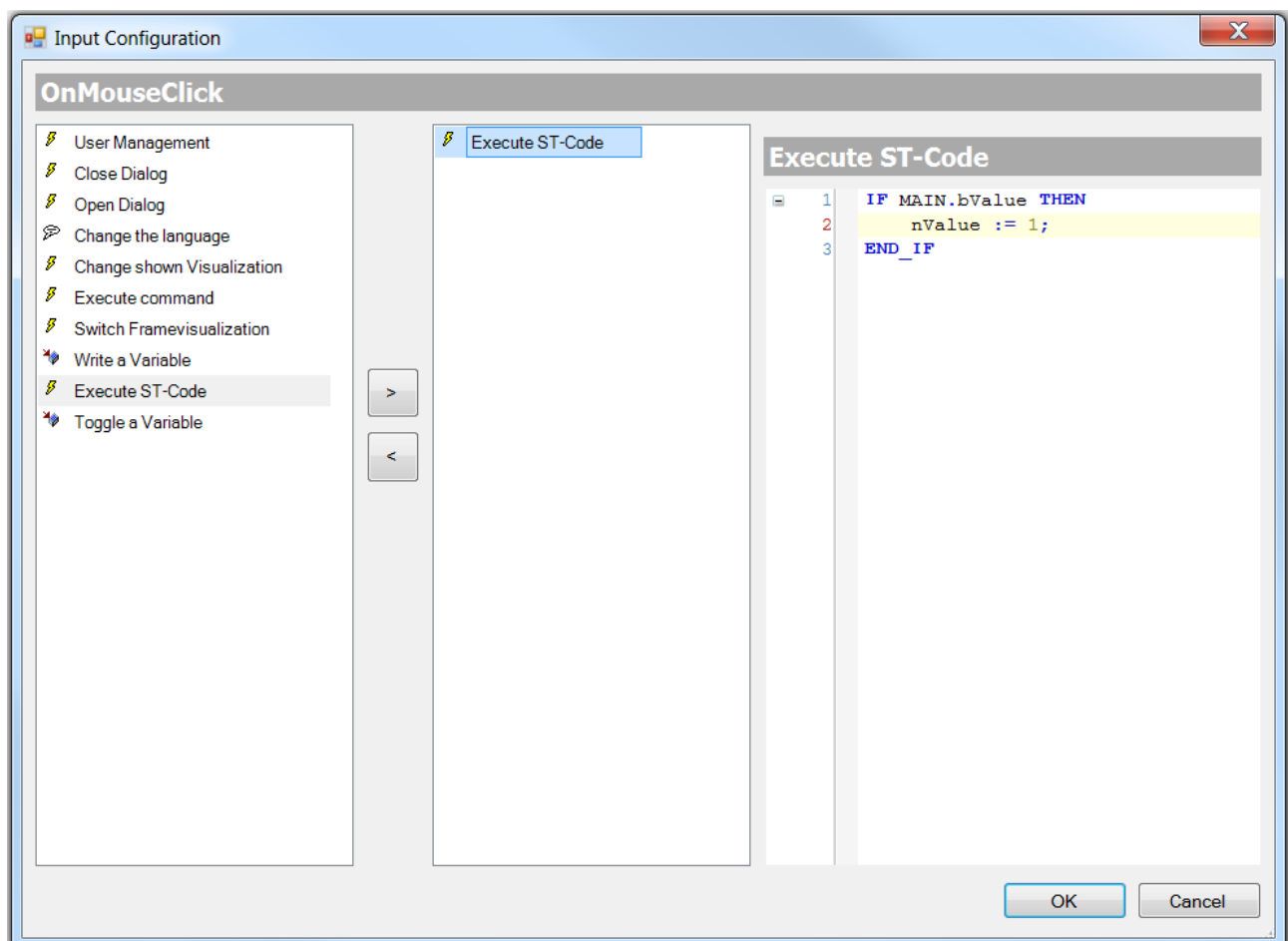
Example

Let's assume the dialog shown above is a configuration dialog for a rectangle element. If the mouse button is pressed on this rectangle in online mode, a numerical keypad appears with the title returned by the variable "MAIN.sTitle".

Click 1, 2 and 3 to enter the value "123", for example. This value is displayed in the upper part of the dialog, also the minimum and maximum values for the entry, which are specified via "MAIN.fMinValue" and "MAIN.fMaxValue". When the entry is confirmed with a mouse click on OK, "123" is written to the variable "MAIN.fValue". If "fValue" is defined as a STRING variable, it is given the value "123". If "fValue" is a numeric variable, it is assigned the value 123.

Execute ST-Code

In the input field for this subsequent action it is possible to enter code in Structured Text, which is to be executed following a mouse action.

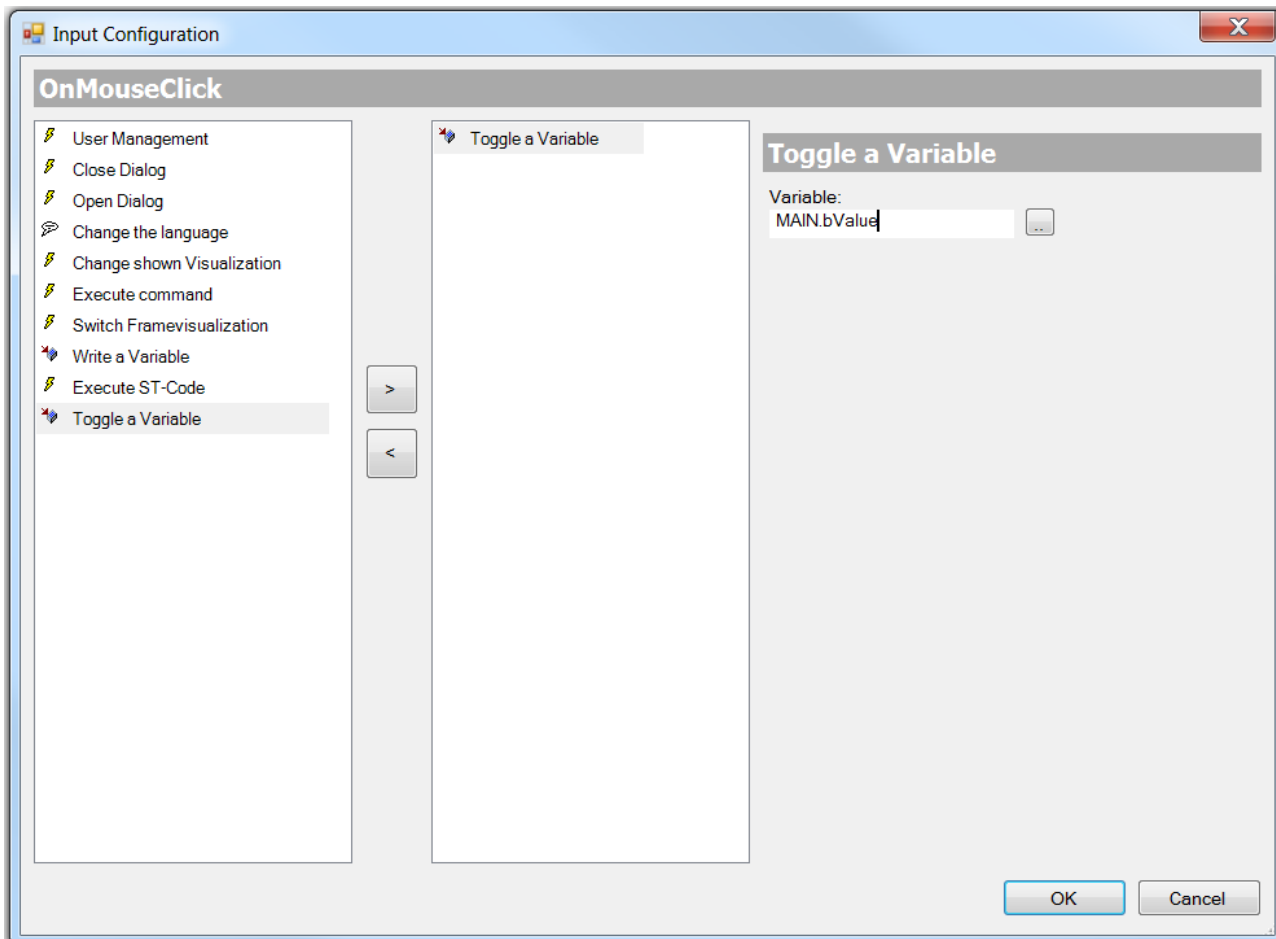




A complex ST program is only possible if the [PLC HMI](#) [▶_603] and/or the [PLC HMI Web](#) [▶_608] is enabled. Otherwise only simple assignments are possible.

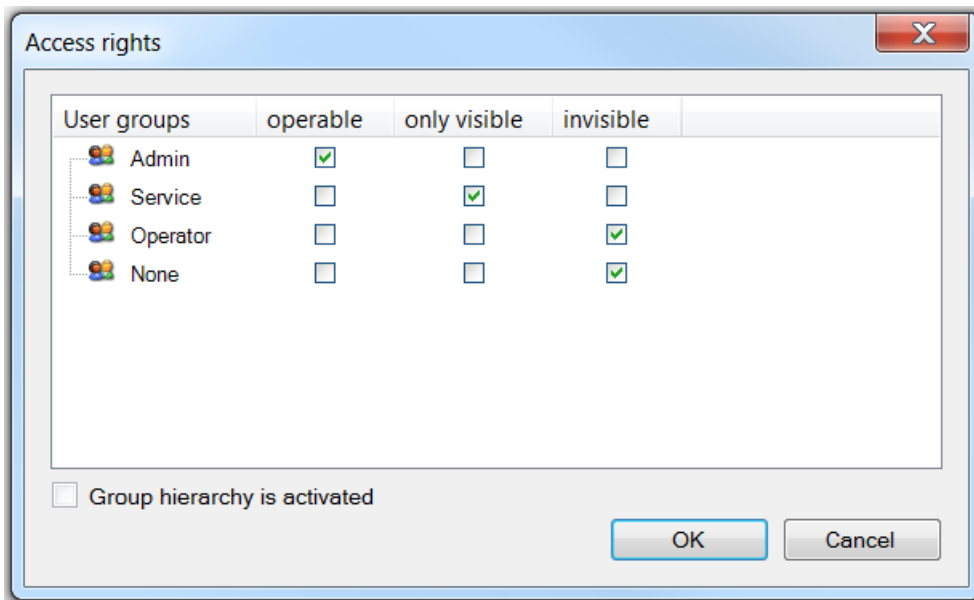
Toggle a Variable

Here you can enter a boolean variable, which is to switch between TRUE and FALSE on repeated mouse actions.



15.8.1.3 Access rights dialog

A mouse click in the value field for the access rights under the [Properties](#) [▶_385] of an element opens the following dialog:



User groups	The groups that were configured in the Visualization Manager under User management → Groups are listed here. Tick a checkbox in the row of the group for which access rights are to be assigned.
operable	If the checkbox is ticked, the element offers full functionality.
only visible	The element offers no functionality, although it is visible
invisible	The element is not displayed for the respective user groups.
Group hierarchy is activated	If the group hierarchy is set to "Use" in the Visualization Manager under User management → Settings, this is displayed here, and the deactivated checkbox is ticked.

Standard group "None"

"None" is the default group and cannot be deleted. If no user is logged in, the visualization elements behave according to the configuration of "None".

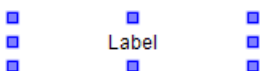


If the access rights of an element are limited, it is recommended to assign only minimum rights to "None".

15.8.2 Common Controls

15.8.2.1 Labelling

The element Label can be used to add labels, headings and any other type of text to a visualization.



Properties editor


The properties of a visualization element - except alignment and order [▶ 377] - can all be configured in the properties editor [▶ 385]. By default, this editor opens next to the visualization editor, or it can be opened explicitly via the "Properties" command (which can be found in the View menu as standard).

A property can be modified by editing the field "Value". To this end, an input field, a selection list, a dialog or checkbox that can be activated is provided in this field, depending on the element type. The value field opens

- after a double-click,
- after a single click in a selected field,

- via the space bar, if the field was already selected.

If a variable is assigned,

- simply enter its name.
- Use the  button to open the input assistant for selecting a variable. The Variables category lists all variables that have already been defined in the project.

Working in the list of properties can be made easier with the aid of default, sorting and filter functions.

Element properties

All element properties and their descriptions are listed below.

Element name	The element name can be changed. Standard name is "GenElemInst_x". "x" stands for a sequential number.
Element type	The element type is entered here. For three element groups it is possible to switch between the corresponding elements by changing the element type: <ul style="list-style-type: none"> • rectangle, rounded rectangle and ellipse [► 475] • Polygon, polyline or Bézier curve [► 490] • dip switch, power switch, push switch, push switch with LED, rocker switch [► 532]

Position

Here you can define the position (X/Y coordinates) and size (width and height) of the element in pixels. The origin is in the top left corner of the window. The positive x-axis is on the right, the positive y-axis runs downwards. If the values edited, the displayed element is simultaneously modified in the [visualization editor \[► 376\]](#).

X	Horizontal position in pixels – X=0 is the left edge of the window.
Y	Vertical position in pixels – Y=0 is the upper edge of the window.
Width	Width of the element in pixels
Height	Height of the element in pixels



Texts

These properties are used for a static definition of the element labelling.

Text	Enter a text. It is used to label the element. The text can also be entered directly. If the element is selected in the visualization editor, an input field can be opened by pressing the space bar.
------	--

Text properties

These properties are used for a static definition of the font. A dynamic definition of the font is possible in the category "Font variables".

Horizontal alignment	Defines the horizontal alignment of the text through selection of: <ul style="list-style-type: none"> • Left • Centered • Right
Vertical alignment	Defines the vertical alignment of the text through selection of: <ul style="list-style-type: none"> • Top • Centered • Bottom
Text format	Defines the display of a text that is too long to be displayed completely in an element: <ul style="list-style-type: none"> • Default – the text extends beyond the element. • Line break – the text is automatically wrapped. • Ellipsis - the text is displayed as far as possible and then truncated with "...".
Font:	Defines the font through selection from predefined fonts: <ul style="list-style-type: none"> • Standard • Heading • Large heading • Title • Note Press  to opens the dialog for user-defined font properties.
Font color	Defines the font color for the element. Either from the selection list or via the dialog that opens when  is clicked.

State variables

These are dynamic definitions of the availability of the element in online mode.

Invisibility	Boolean variable. If this returns TRUE, the element is invisible in online mode.
--------------	--

Access rights

This setting relates to the access rights for the individual element. Click to open the [Access rights dialog](#) [▶ 420]. The setting is only available if a [user management](#) [▶ 393] was added to the PLC project. The following status messages are available:

Not set. All rights.	The default message is set, if the element is shown as available for all groups.
Rights are issued: Limited rights.	The message is set, if the element is shown with limited behavior for at least one group.

15.8.2.2 Integer combo box

The Integer combo box can be used to select a value from a drop-down menu. The index of the select value is written into a variable. The entries can be taken from a text list or as an image list from the image pool.




Properties editor

The properties of a visualization element - except [alignment and order](#) [▶ 377] - can all be configured in the [properties editor](#) [▶ 385]. By default, this editor opens next to the visualization editor, or it can be opened explicitly via the "Properties" command (which can be found in the View menu as standard).

A property can be modified by editing the field "Value". To this end, an input field, a selection list, a dialog or checkbox that can be activated is provided in this field, depending on the element type. The value field opens

- after a double-click,
- after a single click in a selected field,
- via the space bar, if the field was already selected.

If a variable is assigned,

- simply enter its name.
- Use the  button to open the input assistant for selecting a variable. The Variables category lists all variables that have already been defined in the project.

Working in the list of properties can be made easier with the aid of default, sorting and filter functions.

Element properties

All element properties and their descriptions are listed below.

Element name	The element name can be changed. Standard name is "GenElemInst_x". "x" stands for a sequential number.
Element type	The element type is entered here. For three element groups it is possible to switch between the corresponding elements by changing the element type: <ul style="list-style-type: none"> • rectangle, rounded rectangle and ellipse [▶ 475] • Polygon, polyline or Bézier curve [▶ 490] • dip switch, power switch, push switch, push switch with LED, rocker switch [▶ 532]

Position

Here you can define the position (X/Y coordinates) and size (width and height) of the element in pixels. The origin is in the top left corner of the window. The positive x-axis is on the right, the positive y-axis runs downwards. If the values edited, the displayed element is simultaneously modified in the [visualization editor \[▶ 376\]](#).

X	Horizontal position in pixels – X=0 is the left edge of the window.
Y	Vertical position in pixels – Y=0 is the upper edge of the window.
Width	Width of the element in pixels
Height	Height of the element in pixels



Texts

Tooltip	Here you can set the text to be used as tooltip for the element. It only appears in the visualization at runtime.
---------	---

Text properties

Use of	The following two settings are available for selection: <ul style="list-style-type: none"> • Default style values • User-defined settings <ul style="list-style-type: none"> ◦ Horizontal alignment ◦ Vertical alignment ◦ Font ◦ Font color
--------	---

User-defined text properties

Horizontal alignment	Defines the horizontal alignment of the text through selection of: <ul style="list-style-type: none"> • Left • Centered • Right
Vertical alignment	Defines the vertical alignment of the text through selection of: <ul style="list-style-type: none"> • Top • Centered • Bottom
Font:	Defines the font through selection from predefined fonts: <ul style="list-style-type: none"> • Standard • Heading • Large heading • Title • Note Press  to opens the dialog for user-defined font properties.
Font color	Defines the font color for the element. Either from the selection list or via the dialog that opens when  is clicked.

Subsection

Use subrange	If this option is enabled, only a subsection of the assigned list entries is used.
Index start	Variable of type integer, which indexes the entry to be used first
Index end	Variable of type integer, which indexes the entry to be used last
Filter missing textentries	If this option is enabled, only the entries from the text list that actually contain text are displayed.

Access rights

This setting relates to the access rights for the individual element. Click to open the [Access rights dialog \[► 420\]](#). The setting is only available if a [user management \[► 393\]](#) was added to the PLC project. The following status messages are available:

Not set. All rights.	The default message is set, if the element is shown as available for all groups.
Rights are issued: Limited rights.	The message is set, if the element is shown with limited behavior for at least one group.

Additional settings

Variable	Variable of type integer, the value of which is the index of the selected list element.
Text list	Name (string) of the text list that contains and indexes all the entries of the drop down menu.
Image pool	Name (string) of the image pool that contains and indexes all the images of the drop down menu.
Maximum value	Maximum number of entries in the drop down menu

State variables

These are dynamic definitions of the availability of the element in online mode.

Invisibility	Specification of a boolean variable. If this returns TRUE, the element is invisible in online mode.
Input disabled	Specification of a boolean variable. If TRUE is returned, inputs for the element have no effect. Also, the element itself is greyed out in the visualization, to indicate that no user inputs are possible. If the visualization uses the user management [▶ 393] , the elements for user groups with access right "only visible" are grayed out.

15.8.2.3 Combobox Array

The Combobox Array can be used to select a row with values from a drop down menu. The index of the selected row is written into a variable.




Properties editor

The properties of a visualization element - except [alignment and order \[▶ 377\]](#) - can all be configured in the [properties editor \[▶ 385\]](#). By default, this editor opens next to the visualization editor, or it can be opened explicitly via the "Properties" command (which can be found in the View menu as standard).

A property can be modified by editing the field "Value". To this end, an input field, a selection list, a dialog or checkbox that can be activated is provided in this field, depending on the element type. The value field opens

- after a double-click,
- after a single click in a selected field,
- via the space bar, if the field was already selected.

If a variable is assigned,

- simply enter its name.
- Use the  button to open the input assistant for selecting a variable. The Variables category lists all variables that have already been defined in the project.

Working in the list of properties can be made easier with the aid of default, sorting and filter functions.

Element properties

All element properties and their descriptions are listed below.

Element name	The element name can be changed. Standard name is "GenElemInst_x". "x" stands for a sequential number.
Element type	The element type is entered here. For three element groups it is possible to switch between the corresponding elements by changing the element type: <ul style="list-style-type: none"> • rectangle, rounded rectangle and ellipse [▶ 475] • Polygon, polyline or Bézier curve [▶ 490] • dip switch, power switch, push switch, push switch with LED, rocker switch [▶ 532]

Position

Here you can define the position (X/Y coordinates) and size (width and height) of the element in pixels. The origin is in the top left corner of the window. The positive x-axis is on the right, the positive y-axis runs downwards. If the values edited, the displayed element is simultaneously modified in the [visualization editor \[▶ 376\]](#).


X	Horizontal position in pixels – X=0 is the left edge of the window.
Y	Vertical position in pixels – Y=0 is the upper edge of the window.
Width	Width of the element in pixels
Height	Height of the element in pixels

Columns

The Element Combobox Array visualizes an array or structure variable in table view. The index of array elements or components of a structure is shown in a column or a row. Two-dimensional arrays or arrays of a structure displayed in several columns. The variable to be displayed is specified in the property 'Data array'. If a variable is assigned there, the display of the table columns, in which the respective array elements are shown, can be specified. Each column that is assigned an index [<n>] can be configured individually.

Columns • [<n>]	The variable structure defined under "Data array" is used to determine the number of columns and assign the index <n>. Example: <pre>aStringTable : ARRAY [0..2, 0..4] OF STRING := [,BMW', ,Audi', ,Mercedes', ,VW', ,Fiat', ,150', ,150', ,150', ,150', ,100', ,blau', ,grau', ,silber', ,blau', ,rot'</pre> → Three columns are formed [0], [1] and [2].
Max. array index	Numeric variable that is used to specify a maximum array index (optional)
Row height	Row height in pixels
Number of visible rows	This parameter can be used to specify how many rows are to be shown in the drop-down menu. If the number of rows is smaller is than the number of rows in the array, a scrollbar is displayed.
Size of the scrollbar	Width of the vertical scrollbar in pixels

Column [<n>]

Width	Column width in pixels
Image column	If this option is enabled, this column is used to display images from the global image pool or from user-defined image pools. The values in the table cells specify the ID of the image in the image pool.
Image configuration • Fill mode • Transparency • Transparency color	The following configurations can be implemented here: • Fill mode Fill cell: The image is adjusted to the cell size, irrespective of the width/height ratio. Centered: The image is centered while maintaining the proportions within the cell. The width/height ratio is maintained, even if the height or width is adjusted individually. • Transparency If this option is enabled, the color set under transparency color is made transparent. • Transparency color Here you can select a color from a selection list or via a color selection dialog, which can be opened via the  button. The color is shown transparent, if the corresponding transparency option is enabled.
Text alignment in column	Here you can modify the alignment for the column: • Left • Centered • Right



Texts

Tooltip	Here you can set the text to be used as tooltip for the element. It only appears in the visualization at runtime.
---------	---

Text properties

Use of	<p>The following two settings are available for selection:</p> <ul style="list-style-type: none"> • Default style values • User-defined settings <ul style="list-style-type: none"> ◦ Horizontal alignment ◦ Vertical alignment ◦ Font ◦ Font color
--------	--

User-defined text properties

Horizontal alignment	<p>Defines the horizontal alignment of the text through selection of:</p> <ul style="list-style-type: none"> • Left • Centered • Right
Vertical alignment	<p>Defines the vertical alignment of the text through selection of:</p> <ul style="list-style-type: none"> • Top • Centered • Bottom
Font:	<p>Defines the font through selection from predefined fonts:</p> <ul style="list-style-type: none"> • Standard • Heading • Large heading • Title • Note <p>Press  to opens the dialog for user-defined font properties.</p>
Font color	<p>Defines the font color for the element. Either from the selection list or via the dialog that opens when  is clicked.</p>

Access rights

This setting relates to the access rights for the individual element. Click to open the [Access rights dialog](#) [► 420]. The setting is only available if a [user management](#) [► 393] was added to the PLC project. The following status messages are available:

Not set. All rights.	The default message is set, if the element is shown as available for all groups.
Rights are issued: Limited rights.	The message is set, if the element is shown with limited behavior for at least one group.

Additional settings

Variable	Variable of type integer, the value of which is the index of the selected row.
Data array	Variable of the type Array that is to be displayed. The structure of the variable determines the number of columns and rows in the table.

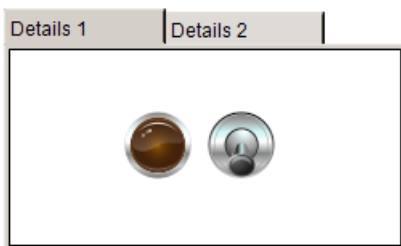
State variables

These are dynamic definitions of the availability of the element in online mode.

Invisibility	Specification of a boolean variable. If this returns TRUE, the element is invisible in online mode.
Input disabled	Specification of a boolean variable. If TRUE is returned, inputs for the element have no effect. Also, the element itself is greyed out in the visualization, to indicate that no user inputs are possible. If the visualization uses the user management [▶ 393], the elements for user groups with access right "only visible" are greyed out.

15.8.2.4 Tab control

A tab control enables several visualization pages to be displayed within a window. Tabs are used to switch between the visualization pages. The "[Toggle variable](#) [[▶ 430](#)]" can be used to specify the page to be displayed from the program code.



If not all tabs can be displayed due to the small width of the element, two navigation arrows will be displayed at the top right on the element.


Properties editor

The properties of a visualization element - except [alignment and order](#) [[▶ 377](#)] - can all be configured in the [properties editor](#) [[▶ 385](#)]. By default, this editor opens next to the visualization editor, or it can be opened explicitly via the "Properties" command (which can be found in the View menu as standard).

A property can be modified by editing the field "Value". To this end, an input field, a selection list, a dialog or checkbox that can be activated is provided in this field, depending on the element type. The value field opens

- after a double-click,
- after a single click in a selected field,
- via the space bar, if the field was already selected.

If a variable is assigned,

- simply enter its name.
- Use the  button to open the input assistant for selecting a variable. The Variables category lists all variables that have already been defined in the project.

Working in the list of properties can be made easier with the aid of default, sorting and filter functions.

Element properties

All element properties and their descriptions are listed below.

Element name	The element name can be changed. Standard name is "GenElemInst_x". "x" stands for a sequential number.
Element type	The element type is entered here. For three element groups it is possible to switch between the corresponding elements by changing the element type: <ul style="list-style-type: none"> • rectangle, rounded rectangle and ellipse [▶ 475] • Polygon, polyline or Bézier curve [▶ 490] • dip switch, power switch, push switch, push switch with LED, rocker switch [▶ 532]

Position

Here you can define the position (X/Y coordinates) and size (width and height) of the element in pixels. The origin is in the top left corner of the window. The positive x-axis is on the right, the positive y-axis runs downwards. If the values edited, the displayed element is simultaneously modified in the [visualization editor \[▶ 376\]](#).

X	Horizontal position in pixels – X=0 is the left edge of the window.
Y	Vertical position in pixels – Y=0 is the upper edge of the window.
Width	Width of the element in pixels
Height	Height of the element in pixels

Referenced visualizations

Here you can open the dialog "[Frame selection \[▶ 525\]](#)", which can be used to select the visualization pages to be referenced. Once one or several visualization pages have been selected, they are listed below with their [placeholders \[▶ 612\]](#), if available. If the placeholders change, the dialog "[Updating the frame parameters \[▶ 526\]](#)" automatically opens for all instances.

Toggle variable

This property can be used to switch between visualizations of a frame.

Variable	Integer variable whose value contains the ID of the visualization to be displayed. The ID of a visualization is determined by the order in the list of assigned visualizations in the Frame selection [▶ 617] . For example, the first entry in this list has the ID 0, the second entry the ID 1. The visualizations, which are assigned to a frame, can be toggled with a variable. The value (ID) of the visualization is determined by the order of this element in the list of the assigned visualizations in the dialog "Configuration of the frame visualizations". For integer values, the first entry in this list results in the value 0, the second entry 1 etc.
----------	--

Access rights

This setting relates to the access rights for the individual element. Click to open the [Access rights dialog \[▶ 420\]](#). The setting is only available if a [user management \[▶ 393\]](#) was added to the PLC project. The following status messages are available:

Not set. All rights.	The default message is set, if the element is shown as available for all groups.
Rights are issued: Limited rights.	The message is set, if the element is shown with limited behavior for at least one group.

Additional settings

Tab width	Tab width in pixels
Scaling type	<p>This parameter is used to specify how the window responds to changes in the size:</p> <ul style="list-style-type: none"> • Isotropic <p>The referenced element maintains its proportions. This means that the ratio between height and width of the visualization remains unchanged, even if the height and width are changed independently.</p> <ul style="list-style-type: none"> • Anisotropic <p>The tab control element follows the size, so that the height and width of the referencing visualization can be changed independently.</p> <ul style="list-style-type: none"> • Unscaled <p>The original size of the visualization is maintained, irrespective of the size of the tab control.</p> <ul style="list-style-type: none"> • Unscaled and scrollable <p>If this option is used, the referenced visualization is shown without scaling. If it is larger than the window area of the frame, the frame is provided with scrollbars, so that the displayed area of the visualization can be moved. To set the position of the scrollbar with a variable, use the properties "Variable scroll position horizontal" and "Variable scroll position vertical".</p>
Deactivate background drawing	To optimize the performance of the visualization, the non-animated elements of the frame element are drawn as a background bitmap. This could result in elements being shown not in the expected order. The function can be disabled to avoid this behavior.

Scrollbar settings

The scrollbar settings are only visible, if the scaling type "Unscaled and scrollable" is entered. It is highly recommended to use the variables on a client-specific basis. In this case, if the variables change, or if a scrollbar is moved with the mouse, the change only affects the frame of the respective client. Otherwise all clients are updated.

Variable scroll position horizontal	Variable containing the position of the horizontal scrollbar.
Variable scroll position vertical	Variable containing the position of the vertical scrollbar.

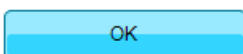
State variables

These are dynamic definitions of the availability of the element in online mode.

Invisibility	Specification of a boolean variable. If this returns TRUE, the element is invisible in online mode.
Input disabled	<p>Specification of a boolean variable. If TRUE is returned, inputs for the element have no effect. Also, the element itself is greyed out in the visualization, to indicate that no user inputs are possible.</p> <p>If the visualization uses the user management [▶ 393], the elements for user groups with access right "only visible" are grayed out.</p>

15.8.2.5 Button

A function assigned to the control element can be triggered with a button. The display can be customized by assigning an [image](#) [[▶ 433](#)] or by configuring an [embossed view](#) [[▶ 433](#)].




Properties editor

The properties of a visualization element - except [alignment and order \[▶ 377\]](#) - can all be configured in the [properties editor \[▶ 385\]](#). By default, this editor opens next to the visualization editor, or it can be opened explicitly via the "Properties" command (which can be found in the View menu as standard).

A property can be modified by editing the field "Value". To this end, an input field, a selection list, a dialog or checkbox that can be activated is provided in this field, depending on the element type. The value field opens

- after a double-click,
- after a single click in a selected field,
- via the space bar, if the field was already selected.

If a variable is assigned,

- simply enter its name.
- Use the  button to open the input assistant for selecting a variable. The Variables category lists all variables that have already been defined in the project.

Working in the list of properties can be made easier with the aid of default, sorting and filter functions.

Element properties

All element properties and their descriptions are listed below.


Element name	The element name can be changed. Standard name is "GenElemInst_x". "x" stands for a sequential number.
Element type	The element type is entered here. For three element groups it is possible to switch between the corresponding elements by changing the element type: <ul style="list-style-type: none"> • rectangle, rounded rectangle and ellipse [▶ 475] • Polygon, polyline or Bézier curve [▶ 490] • dip switch, power switch, push switch, push switch with LED, rocker switch [▶ 532]

Position

Here you can define the position (X/Y coordinates) and size (width and height) of the element in pixels. The origin is in the top left corner of the window. The positive x-axis is on the right, the positive y-axis runs downwards. If the values edited, the displayed element is simultaneously modified in the [visualization editor \[▶ 376\]](#).

X	Horizontal position in pixels – X=0 is the left edge of the window.
Y	Vertical position in pixels – Y=0 is the upper edge of the window.
Width	Width of the element in pixels
Height	Height of the element in pixels

Center

When a value is edited, the corresponding element  is simultaneous moved in the [visualization editor \[▶ 376\]](#).

X	Horizontal position of the element pivot in pixels
Y	Vertical position of the element pivot in pixels

Colors

A color is defined based on a hexadecimal number consisting of red / green / blue (RGB) components. For each of these three colors, 256 (0-255) values are available, which can be entered here statically. The color can be selected from a selection list or via the color selection dialog, which can be opened via the button




. In addition, the level of transparency can be set for each color (0: fully transparent, 255: fully opaque).

Color	Select a frame and fill color for the default state. If the color switching variable is defined as FALSE, the element is in default state.
Alarm color	Select a frame and fill color for the element in alarm state. This is triggered, if the color switching variable is defined as TRUE.
Use color gradient	The checkbox is unticked by default. If it is ticked, the corresponding element is drawn with a color gradient.
Color gradient selection	The Gradient editor [▶ 405] opens.
Button height	Height in pixels for the button in embossed view



Color gradient and transparency are not supported under Windows CE.

Image



Static ID	Static ID of the image file for the button. The ID (string) is defined in the image pool. The name of the image pool should be prefixed in order to make the entry unambiguous. For image files that are managed in the GlobalImagePool, it is not necessary to specify an image pool, since this image pool is always searched first. Click the  button to open the input assistant with a list of all available image pools and their images.
Scaling type	Here you can specify how the image file responds to changes in the button size: <ul style="list-style-type: none"> • Isotropic: The image retains its proportions; i.e. the height/width ratio is maintained, even if the height and width of the button are changed separately. • Anisotropic: The image retains its proportions; i.e. the height/width ratio is maintained, even if the height and width of the button are changed separately. • Unscaled: The image retains its original size, even if the size of the button changes.
Transparency	If this option is enabled, the bitmap is made transparent with the color set under "Transparency color".
Transparency color	Here you can set the color to be shown fully transparent in the image. Requirement: Transparency is enabled
Horizontal alignment	Here you can define the horizontal alignment of the bitmap: <ul style="list-style-type: none"> • Left • Centered • Right
Vertical alignment	Here you can define the vertical alignment of the bitmap: <ul style="list-style-type: none"> • Top • Centered • Bottom

Texts

These properties are used for a static definition of the element labelling. Each can contain a formatting sequence [▶ 617], e.g. %s. In online mode the sequence is replaced by the contents of the variables defined in "Text variables".




Text	Enter a text. It is used to label the element. The text can contain a formatting sequence, e.g. %s. The corresponding variable is defined in "Text variables". Note: The text can also be entered directly. If the element is selected in the visualization editor, an input field can be opened by pressing the space bar.
Tooltip	Enter a text. It is used as a tooltip for the element and appears in the visualization only in online mode when the cursor is placed over an element. The text can contain a formatting sequence, e.g. %s. The corresponding variable is defined in "Text variables".

Text properties

Font	Defines the font through selection from predefined fonts: <ul style="list-style-type: none"> • Standard • Heading • Large heading • Title • Note Press  to opens the dialog for user-defined font properties.
Font color	Defines the font color for the element. Either from the selection list or via the dialog that opens when  is clicked.

Absolute movement

The element can be moved by changing the x- and y-position (pixels) of the top left corner of the element via an integer variable. Absolute coordinate values are used here.

Motion <ul style="list-style-type: none"> • X • Y 	<p>X: The integer variable entered here defines the current x-position of the top left corner of the element (in pixels). It can be used to move the element in x-direction. (A positive value moves the element from left to right).</p> <p>Y: The integer variable entered here defines the current y-position of the top left corner of the element (in pixels). It can be used to move the element in y-direction. (A positive value moves the element from top to bottom).</p>
Rotation	<p>The integer variable entered here defines the angle (angular degrees) for rotating the element around a rotation point.</p> <p>Positive values = clockwise</p> <p>Note: In contrast to the behavior with 'internal rotation' (see below), the element itself does not rotate. Click on the element to show the rotation point . It can be moved by pressing and holding the mouse button.</p>
Scaling	<p>The integer variable entered here defines the current scaling factor (percent). The element size is adjusted linearly according to this value. The value is implicitly divides by 1000, so that it is not necessary to use REAL variables in order to shrink the element. The scaling always refers to the rotation point (center). Click on the element to show the rotation point . It can be moved by pressing and holding the mouse button.</p>
Internal rotation	<p>The integer variable entered here defines the angle (angular degrees) by which the element is rotated around its rotation point; positive values=mathematically positive=clockwise. In contrast to "Rotation" (see above), the element itself rotates. Click on the element to show the rotation point (center) . It can be moved by pressing and holding the mouse button.</p>

Relative movement

The element can be moved relative to its fixed position. The top left and bottom right edges of the element are moved in x- or y-direction by a value (pixels) defined by an integer variable. In contrast to an absolute movement, a relative position is defined, i.e. the distance to the original position. This function can be used to change the shape of the element. Positive values move the horizontal edges downwards and/or the vertical edges to the right.

Top left movement • X • Y	<ul style="list-style-type: none"> • X: Integer variable, the value of which indicates the number of pixels by which the top left corner is moved in x-direction. • Y: Integer variable, the value of which indicates the number of pixels by which the top left corner is moved in y-direction.
Bottom right movement • X • Y	<ul style="list-style-type: none"> • X: Integer variable, the value of which indicates the number of pixels by which the bottom right corner is moved in x-direction. • Y: Integer variable, the value of which indicates the number of pixels by which the bottom right corner is moved in y-direction.

Text variables

You can display text that is stored in a variable. To do this, first add a formatting sequence in the text that is defined under the "Texts" property. Then assign a variable. In online mode the [formatting sequence](#) [▶ 617] is replaced by the contents of the variables.

Example %f:

Enter "Result: %2.5f" in the "Texts" property. Enter "fValue" in the "Text variables" property. The variable must be a defined IEC variable. The element will then be labeled in online mode with "Result: 12.12345" if fValue = 12.1234567.

Text variable	Variable (of default data type) containing the information to be displayed. The type must match the formatting sequence in the "Texts" setting.
Tooltipvariable	Variable of type String containing the tooltip text to be displayed. The entry in the property "Texts" must contain a formatting sequence.

Dynamic texts

These parameters are used to define dynamic texts originating from [text lists](#) [▶ 138]. This enables [language change](#) [▶ 616], for example.

A further possibility to define a text dynamically is to supply the text via a string variable. (see the category "Text variables")

Text list	Name of the text list, as used in the project tree, as a string Example: 'TL_ErrorList'
Textindex	Index (ID) of the text, as defined in the text list, as a string. It can be specified directly in a static manner or as a string variable.
Tooltipindex	Index (ID) of the tooltip text, as defined in the text list, as a string. It can be specified directly in a static manner or as a string variable.

Font variables

These variables are used for dynamic font definitions for element texts via project variables. Static definitions are configured under "Text properties".

Font name	Specification of a variable of the type String that contains the font name that is to be used to label the element. (name specified as in the standard font dialog) Example: MAIN.sFont (sFont := 'Arial');
Size	Variable of type INT containing the size of the element text in pixels, as in the default dialog 'Font'. Example: MAIN.nHeight (nHeight := 16;)
Flags	Variable of type DWORD for defining the font display via one of the flag values listed below. A combined definition can be achieved by adding the respective flag values and specifying the sum. <ul style="list-style-type: none"> • 1: italics • 2: bold • 4: underlined • 8: deleted Example: MAIN.nFlag (nFlag := 6;) The text is shown bold and underlined.
Character set	The character set to be used for the font can be defined via the default font number. This number can be specified via a DWORD variable (see also the definition in the default font dialog)
Color	Variable of type DWORD for defining the color of the element text.
Flags for the text alignment	Variable of type DWORD for defining the text alignment via one of the flag values listed below. A combined definition can be achieved by adding the respective flag values and specifying the sum. <ul style="list-style-type: none"> • 0: top left • 1: centered horizontally • 2: right • 4: centered vertically • 8: below Example MAIN.nFlag (nFlag := 5;) The text is displayed centered horizontally and vertically.

Color variables

The color variables used for dynamic definition of the element colors via project variables of type DWORD. A color is defined based on a hexadecimal number consisting of red, green and blue (RGB) components. In addition, the variables are used to specify the transparency of the color (FF: fully opaque - 00: fully transparent). The DWORD has the following structure: 16#TTRRGGBB



The structure of the hexadecimal number differs from TwinCAT 2. In TwinCAT 3 it is possible to define the transparency of the color with the hexadecimal number in addition to the RGB proportions. The transparency is indicated by the first two digits after "16#". Colors with definitions starting with "16#00" are invisible, since they are fully transparent.

Example:

```
nFillColor := 16#FF8FE03F;
```

- FF: transparency (fully opaque)
- 8F: red
- E0: green
- 3F: blue

Color change	Boolean variable, which controls the switching of the element color between "normal state" (variable = FALSE) and "alarm state" (variable = TRUE).
Normal state	Variable of type DWORD for defining the element color. It overwrites the value that is currently defined in "Colors". The value in the project variable is used, if the variable defined in "Color change" is FALSE.
Alarm state	Variable of type DWORD for defining the element color in alarm state. It overwrites the value that is currently defined in "Colors". The value in the project variable is used, if the variable defined in "Color change" is TRUE.



Transparency is not supported under Windows CE.

State variables

These are dynamic definitions of the availability of the element in online mode.

Invisibility	Specification of a boolean variable. If this returns TRUE, the element is invisible in online mode.
Input disabled	Specification of a boolean variable. If TRUE is returned, inputs for the element have no effect. Also, the element itself is greyed out in the visualization, to indicate that no user inputs are possible. If the visualization uses the user management [▶ 393], the elements for user groups with access right "only visible" are grayed out.

Button state variable

Boolean value	Boolean variable, which controls the button display as "pressed" if the variable is TRUE, or as "not pressed" if the variable is FALSE.
---------------	---

Image ID variable

Image ID	Project variable of the type String that contains the image ID with which the element is to be displayed. The ID is defined in the image pool [▶ 146]. The name of the image pool should be prefixed in order to make the entry unambiguous. For image files that are managed in the "GlobalImagePool", it is not necessary to specify an image pool, since this image pool is always searched first.
----------	---

Input configuration

Here you can define the consequential action that should be performed when the user makes an input in the element in online mode. As long as no follow-on actions are defined, "Configure..." appears in the Properties field. Click on "Configure..." to open the [Input Configuration](#) [[▶ 406](#)], where you can assign follow-on actions. Each input action can be assigned any number of follow-on actions.

The following input events are available for an element:

- OnDialogClosed
- OnMouseClicked
- OnMouseDown
- OnMouseEnter
- OnMouseLeave
- OnMouseMove
- OnMouseUp

OnDialogClosed	This event is triggered if one of the dialog boxes that were opened for user inputs is closed within a visualization. Note: This property is not limited to the element, for which it is configured, but applies within the entire visualization. It therefore does not respond to each dialog closing action. Currently there is no way to define such a property for the entire visualization. It therefore has to be assigned to one of its elements.
OnMouseClicked	This mouse event is triggered when the cursor points to an element and a full mouse click (pressing and releasing the mouse button) is executed on this item.
OnMouseDown	This mouse event is triggered when the mouse button is pressed while the cursor points to an element. It is irrelevant where on the visualization the mouse button is released again.
OnMouseEnter	This mouse event is triggered when the cursor is dragged to the element. It is irrelevant whether the mouse button is pressed or released.
OnMouseLeave	This mouse event is triggered when the cursor leaves the element. It is irrelevant whether the mouse button is pressed or released.
OnMouseMove	This mouse event is triggered when the cursor is moved within the element. It is irrelevant whether the mouse button is pressed or released.
OnMouseUp	This mouse event occurs when the mouse button is released on the element. The mouse button was pressed prior to that outside of the element.

Input configuration – Keys

"Keys" can be used to specify that the value of a boolean project variable is set depending on the mouse behavior for the event "Keys".

Variable	Boolean variable whose value is TRUE if the mouse button is pressed while the cursor points to the element. The value becomes FALSE again when the mouse button is released or the cursor leaves the element.
FALSE keys	If this option is enabled, the key behavior described above is reversed for the corresponding variable. That is, when the mouse button pressed, the variable value is set to FALSE. When the button is, the variable value is set to TRUE.
Switch on enter if mouse trapped	If this option is enabled, the variable value behaves as described under 'Variables', as long as the cursor is within the element area. If the cursor leaves the element area while the mouse button is pressed, the variable is set to FALSE. However, it is automatically set to TRUE again, if the cursor returns to the element area without the mouse button being released. In other words, the system takes into account that the mouse is "trapped" as long as the mouse button is pressed, even if the cursor leaves the element area.

Input configuration - Switching

"Switching" can be used to specify that for this event the value of a boolean project variable is set depending on the mouse behavior.

Variable	Boolean variable whose value switches between TRUE and FALSE with each mouse click on the element. No switching occurs if the cursor is moved away from the element while the mouse button is pressed. Can be used to cancel a switching action.
Switch on release, if mouse trapped	If this option is enabled, the variable value behaves as described under 'Variables', as long as the cursor is within the element area. If the cursor leaves the element area while the mouse button is pressed and the mouse button is then released, the variable is switched anyway. In other words, the system takes into account that the mouse is "trapped" as long as the mouse button is pressed, even if the cursor leaves the element area.

Input configuration - hotkeys

A hotkey can be used to define a key or key combination and link it with a follow-on action (e.g. MouseDown, MouseUp), which is to be executed when a key event occurs (KeyDown, KeyUp). By default the MouseDown action is executed on KeyDown (press key) and the MouseUp action on KeyUp (release key). This can be useful if a visualization is to be operated both via mouse actions and keyboard entries, since the input

actions then only have to be configured once. The key configuration for an element is also managed in the [hotkeys configuration](#) [[▶ 392](#)] for the visualization. Any changes are always synchronized between this and the element properties editor.

Key	Assigning a key. A selection list contains all currently supported keys, e.g. M.
Event(s)	Definition of the event to be executed, if the key or key and modifier are used. Possible values that are available in a selection list: <ul style="list-style-type: none"> • No action • MouseDown action when the key is pressed • MouseUp action when the key is released • MouseDown/MouseUp action when the key is pressed/released
Shift	If this option is enabled, the key has to be used in combination with the Shift key.
Control	If this option is enabled, the key has to be used in combination with the Ctrl key.
Alt	If this option is enabled, the key has to be used in combination with the Alt key.

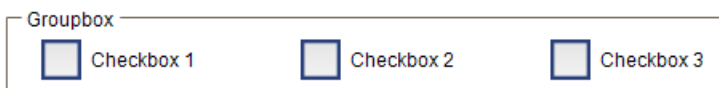
Access rights

This setting relates to the access rights for the individual element. Click to open the [Access rights dialog](#) [[▶ 420](#)]. The setting is only available if a [user management](#) [[▶ 393](#)] was added to the PLC project. The following status messages are available:

Not set. All rights.	The default message is set, if the element is shown as available for all groups.
Rights are issued: Limited rights.	The message is set, if the element is shown with limited behavior for at least one group.

15.8.2.6 Groupbox

Visualization elements can be dragged into a group box based on the drag & drop principle, in order to form a visual unit. They can be nested multiple times.




Properties editor

The properties of a visualization element - except [alignment and order](#) [[▶ 377](#)] - can all be configured in the [properties editor](#) [[▶ 385](#)]. By default, this editor opens next to the visualization editor, or it can be opened explicitly via the "Properties" command (which can be found in the View menu as standard).

A property can be modified by editing the field "Value". To this end, an input field, a selection list, a dialog or checkbox that can be activated is provided in this field, depending on the element type. The value field opens

- after a double-click,
- after a single click in a selected field,
- via the space bar, if the field was already selected.

If a variable is assigned,

- simply enter its name.
- Use the  button to open the input assistant for selecting a variable. The Variables category lists all variables that have already been defined in the project.

Working in the list of properties can be made easier with the aid of default, sorting and filter functions.

Element properties

All element properties and their descriptions are listed below.

Element name	The element name can be changed. Standard name is "GenElemInst_x". "x" stands for a sequential number.
Element type	The element type is entered here. For three element groups it is possible to switch between the corresponding elements by changing the element type: <ul style="list-style-type: none"> • rectangle, rounded rectangle and ellipse [► 475] • Polygon, polyline or Bézier curve [► 490] • dip switch, power switch, push switch, push switch with LED, rocker switch [► 532]

Position

Here you can define the position (X/Y coordinates) and size (width and height) of the element in pixels. The origin is in the top left corner of the window. The positive x-axis is on the right, the positive y-axis runs downwards. If the values edited, the displayed element is simultaneously modified in the [visualization editor \[► 376\]](#).

X	Horizontal position in pixels – X=0 is the left edge of the window.
Y	Vertical position in pixels – Y=0 is the upper edge of the window.
Width	Width of the element in pixels
Height	Height of the element in pixels

Texts

These properties are used for a static definition of the element labelling.

Text	Enter a text. It is used to label the element. Note: The text can also be entered directly. If the element is selected in the visualization editor, an input field can be opened by pressing the space bar.
Tooltip	Enter a text. It is used as a tooltip for the element and appears in the visualization only in online mode when the cursor is placed over an element.

Text properties

Font	Defines the font through selection from predefined fonts: <ul style="list-style-type: none"> • Standard • Heading • Large heading • Title • Note Press <input type="button" value="..."/> to opens the dialog for user-defined font properties.
Font color	Defines the font color for the element. Either from the selection list or via the dialog that opens when <input type="button" value="..."/> is clicked.

Access rights

This setting relates to the access rights for the individual element. Click to open the [Access rights dialog \[► 420\]](#). The setting is only available if a [user management \[► 393\]](#) was added to the PLC project. The following status messages are available:

Not set. All rights.	The default message is set, if the element is shown as available for all groups.
Rights are issued: Limited rights.	The message is set, if the element is shown with limited behavior for at least one group.

State variables

These are dynamic definitions of the availability of the element in online mode.

Invisibility	Specification of a boolean variable. If this returns TRUE, the element is invisible in online mode.
Input disabled	Specification of a boolean variable. If TRUE is returned, inputs for the element have no effect. Also, the element itself is greyed out in the visualization, to indicate that no user inputs are possible. If the visualization uses the user management [▶ 393], the elements for user groups with access right "only visible" are grayed out.

15.8.2.7 Table

A table can be added in the visualization, in order to display one- or two-dimensional arrays, structures or local variables of a POU. An example of the configuration of a table can be found in the section "[Configuration of a table](#) [[▶ 446](#)]".

	X	Y	Vel
1			
2			
3			
4			

Properties editor

The properties of a visualization element - except [alignment and order](#) [[▶ 377](#)] - can all be configured in the [properties editor](#) [[▶ 385](#)]. By default, this editor opens next to the visualization editor, or it can be opened explicitly via the "Properties" command (which can be found in the View menu as standard).

A property can be modified by editing the field "Value". To this end, an input field, a selection list, a dialog or checkbox that can be activated is provided in this field, depending on the element type. The value field opens

- after a double-click,
- after a single click in a selected field,
- via the space bar, if the field was already selected.

If a variable is assigned,

- simply enter its name.
- Use the button to open the input assistant for selecting a variable. The Variables category lists all variables that have already been defined in the project.

Working in the list of properties can be made easier with the aid of default, sorting and filter functions.

Element properties

All element properties and their descriptions are listed below.

Element name	The element name can be changed. Standard name is "GenElemInst_x". "x" stands for a sequential number.
Element type	The element type is entered here. For three element groups it is possible to switch between the corresponding elements by changing the element type: <ul style="list-style-type: none"> • rectangle, rounded rectangle and ellipse [▶ 475] • Polygon, polyline or Bézier curve [▶ 490] • dip switch, power switch, push switch, push switch with LED, rocker switch [▶ 532]

Additional settings

Data array	<p>This parameter is used to specify the variable to be visualize, complete with the full path. The structure of the variable determines the number of columns and rows in the table. Use "Max. Array-Index" if the number of array elements for the variable varies. To update the table, if the size of the array or structure variables has changed, place the cursor into the data array value field and press the Enter key.</p> <p>Example: The following array forms a table with three columns and four rows.</p> <pre>aData : ARRAY [1..3,1..4] OF INT;</pre>
Max. array index	Numeric variable, which dynamically specifies the maximum array index, in order to avoid the array limits being exceeded




If the data array value is changed, i.e. if a new variable is assigned, the other table properties are reset to their default values.

Columns

The table element visualizes a variable in table view. The index of the array/component of the structure is shown in a column/row. For two-dimensional arrays or an array of structures the visualization uses several columns. In these element settings the appearance of the table columns is defined, in which the values of the individual array elements/ structure components/ variables are displayed. Each column pertaining to a certain index can be configured individually.

Columns • [<n>]	<p>The variable structure defined under 'Data array' is used to determine the number of columns and assign the index <n>.</p> <p>Example: <pre>adataArray1 : ARRAY [2..8] OF INT;</pre> → A column [0] is formed. <pre>adataArray2 : ARRAY [1..3, 1..10] OF INT;</pre> → Three columns are formed [0], [1] and [2].</p>
Show row header	If this option is enabled, the row labelling is displayed in the form of assigned row indices.
Show column header	If this option is enabled, the column labelling is displayed in the form of assigned row indices.
Row height	Row height in pixels
Width of row header	Width of the column containing the row header, in pixels
Size of the scrollbar	Width of the vertical or the height of the horizontal scrollbar in pixels

Column [<n>**]**

Column heading	Here you can change the column heading by entering a new title. By default the name of the array or structure with the index or structure component pertaining to the column is used as heading.
Width	Column width in pixels
Image column	If this option is enabled, this column is used to display images from the global image pool or from user-defined image pools. The values in the table cells specify the ID of the image in the image pool.
Image configuration <ul style="list-style-type: none"> • Fill mode • Transparency • Transparency color 	<p>The following configurations can be implemented here:</p> <ul style="list-style-type: none"> • Fill mode <p>Fill cell: The image is adjusted to the cell size, irrespective of the width/height ratio.</p> <p>Centered: The image is centered while maintaining the proportions within the cell. The width/height ratio is maintained, even if the height or width is adjusted individually.</p> <ul style="list-style-type: none"> • Transparency <p>If this option is enabled, the color set under transparency color is made transparent.</p> <ul style="list-style-type: none"> • Transparency color <p>Here you can select a color from a selection list or via a color selection dialog, which can be opened via the  button. The color is shown transparent, if the corresponding transparency option is enabled.</p>
Text alignment of the heading	<p>Here you can modify the alignment of the column header:</p> <ul style="list-style-type: none"> • Left • Centered • Right
Use template	If this option is enabled, a further visualization element (of type rectangle , rounded rectangle or ellipse [► 475]) is added in each cell of this table column. The property list is automatically extended with the properties of this element from the template.
Text alignment of the heading from the template	<p>If this option is enabled, the settings for font (size) and alignment from the inserted template are also applied to the column heading.</p> <p>Enabling this option only takes effect if 'Use template' is enabled at the same time.</p>
Template	<p>Under Template the properties of all elements assigned to the column are listed successively and can be edited as described under rectangle, rounded rectangle or ellipse [► 475].</p> <p>This entry only exists if "Use template" is enabled.</p>



Position

Here you can define the position (X/Y coordinates) and size (width and height) of the element in pixels. The origin is in the top left corner of the window. The positive x-axis is on the right, the positive y-axis runs downwards. If the values edited, the displayed element is simultaneously modified in the [visualization editor \[► 376\]](#).

X	Horizontal position in pixels – X=0 is the left edge of the window.
Y	Vertical position in pixels – Y=0 is the upper edge of the window.
Width	Width of the element in pixels
Height	Height of the element in pixels

Text properties

These properties are used for a static definition of the font. A dynamic definition of the font is possible in the category "Font variables".

Horizontal alignment	<p>Defines the horizontal alignment of the text through selection of:</p> <ul style="list-style-type: none"> • Left • Centered • Right
Vertical alignment	<p>Defines the vertical alignment of the text through selection of:</p> <ul style="list-style-type: none"> • Top • Centered • Bottom
Text format	<p>Defines the display of a text that is too long to be displayed completely in an element:</p> <ul style="list-style-type: none"> • Default – the text extends beyond the element. • Line break – the text is automatically wrapped. • Ellipsis - the text is displayed as far as possible and then truncated with "...".
Font:	<p>Defines the font through selection from predefined fonts:</p> <ul style="list-style-type: none"> • Standard • Heading • Large heading • Title • Note <p>Press  to opens the dialog for user-defined font properties.</p>
Font color	<p>Defines the font color for the element. Either from the selection list or via the dialog that opens when  is clicked.</p>

Dynamic texts

These parameters are used to define dynamic texts originating from [text lists \[▶ 138\]](#). This enables [language change \[▶ 616\]](#), for example.

A further possibility to define a text dynamically is to supply the text via a string variable. (see the category "Text variables")

Text list	<p>Name of the text list, as used in the project tree, as a string</p> <p>Example: 'TL_ErrorList'</p>
Textindex	<p>Index (ID) of the text, as defined in the text list, as a string. It can be specified directly in a static manner or as a string variable.</p>
Tooltipindex	<p>Index (ID) of the tooltip text, as defined in the text list, as a string. It can be specified directly in a static manner or as a string variable.</p>

Font variables

These variables are used for dynamic font definitions for element texts via project variables. Static definitions are configured under "Text properties".

Font name	Specification of a variable of the type String that contains the font name that is to be used to label the element. (name specified as in the standard font dialog) Example: MAIN.sFont (sFont := 'Arial');
Size	Variable of type INT containing the size of the element text in pixels, as in the default dialog 'Font'. Example: MAIN.nHeight (nHeight := 16;)
Flags	Variable of type DWORD for defining the font display via one of the flag values listed below. A combined definition can be achieved by adding the respective flag values and specifying the sum. <ul style="list-style-type: none"> • 1: italics • 2: bold • 4: underlined • 8: deleted Example: MAIN.nFlag (nFlag := 6;) The text is shown bold and underlined.
Character set	The character set to be used for the font can be defined via the default font number. This number can be specified via a DWORD variable (see also the definition in the default font dialog)
Color	Variable of type DWORD for defining the color of the element text.
Flags for the text alignment	Variable of type DWORD for defining the text alignment via one of the flag values listed below. A combined definition can be achieved by adding the respective flag values and specifying the sum. <ul style="list-style-type: none"> • 0: top left • 1: centered horizontally • 2: right • 4: centered vertically • 8: below Example MAIN.nFlag (nFlag := 5;) The text is displayed centered horizontally and vertically.

State variables

These are dynamic definitions of the availability of the element in online mode.

Invisibility	Specification of a boolean variable. If this returns TRUE, the element is invisible in online mode.
Input disabled	Specification of a boolean variable. If TRUE is returned, inputs for the element have no effect. Also, the element itself is greyed out in the visualization, to indicate that no user inputs are possible. If the visualization uses the <u>user management</u> [▶ 393], the elements for user groups with access right "only visible" are grayed out.

Selection

Selection color	Fill color for selected table cells
Selection type	This parameter is used to define how the selection is made when a table row is clicked: <ul style="list-style-type: none"> • No selection • Cell selection: only the cell that was clicked is selected. • Row selection: The row containing the cell that was clicked is selected. • Column selection: The column containing the cell that was clicked is selected. • Row and column selection: The row and column containing the cell that was clicked are selected.
Frame around selected cells	If this option is selected, a frame is drawn around the selected cells.
Variable for column selection	Variable of the type Integer in which the index of the column of the selected cell is stored. In case the data array points to a structure, the structure components are indexed, beginning with 0. Note that this index only then represents the correct position in the array, if no columns were excluded from being shown in the table.
Variable for row selection	Variable of type integer, in which the index of the row of the selected cell is stored
Variable for validity of the column selection	Boolean variable, which is TRUE, if the variable for the column selection contains a valid value
Variable for validity of the row selection	Boolean variable, which is TRUE, if the variable for the column selection contains a valid value

Access rights

This setting relates to the access rights for the individual element. Click to open the [Access rights dialog](#) [► 420]. The setting is only available if a [user management](#) [► 393] was added to the PLC project. The following status messages are available:

Not set. All rights.	The default message is set, if the element is shown as available for all groups.
Rights are issued: Limited rights.	The message is set, if the element is shown with limited behavior for at least one group.

15.8.2.7.1 Configuring a table

First, the variable to be visualized in the variable has to be specified. To this end, enter a variable with full path in "data array". The structure of the variable determines the number of rows and columns in the table. For two-dimensional arrays, the first index determines the number of columns, the second index the number of rows. For a structure, the individual components represent the structure of the columns:

- Example 1:

```
aDim1 : ARRAY [2..5] OF INT;
```

	MAIN.aDim1[INDEX]
2	
3	
4	
5	

- Example 2:

```
aDim2 : ARRAY [0..2, 0..3] OF INT;
```

	MAIN.aDim2[0,INDEX]	MAIN.aDim2[1,INDEX]	MAIN.aDim2[2,INDEX]
0			
1			
2			
3			

• Example 3:

```
TYPE ST_ProductInformation :
STRUCT
  nOrderInPieces : INT;
  bStocked : BOOL;
  nArcticleNumber : INT;
END_STRUCT
END_TYPE

stProductInformation : ST_ProductInformation;
```

	MAIN.stProductInformation.nOrderInPieces	MAIN.stProductInformation.bStocked	MAIN.stProductInformation.nArcticleNumber
0			

• Example 4:

```
aProductInformation : ARRAY [0..3] OF ST_ProductInformation;
```

	MAIN.aProductInformation[INDEX].nOrderInPieces	MAIN.aProductInformation[INDEX].bStocked	MAIN.aProductInformation[INDEX].nArcticleNumber
0			
1			
2			
3			

• Example 5:

```
fbTimer : TON;
```

	MAIN.fbTimer.IN	MAIN.fbTimer.PT	MAIN.fbTimer.Q	MAIN.fbTimer.ET	MAIN.fbTimer.M	MAIN.fbTimer.StartTime
0						

• Example 6:

```
aTimers : ARRAY [0..3] OF TON;
```

	MAIN.aTimers[INDEX].IN	MAIN.aTimers[INDEX].PT	MAIN.aTimers[INDEX].Q	MAIN.aTimers[INDEX].ET	MAIN.aTimers[INDEX].M	MAIN.aTimers[INDEX].StartTime
0						
1						
2						
3						




If the entry in the field "data array" is changed, all other settings for the element properties are reset to their defaults!

Example 4 is used below.

Once a variable has been entered in "data array", the properties for each individual column can be changed. First, the column headers are changed to make them more meaningful. Next, the column width and the header alignment are changed.

Two options are available for changing the column width:


1. In the element properties, under the respective column the value can be changed in the property "Width".
2. Directly in the table elements, the mouse can be moved to the line between two columns. A cursor icon appears , which indicates that the size of the column can now be changed while the mouse button is pressed.

	Order in pieces	Stocked?	Articel number
0			
1			
2			
3			

The row height is set to 25 pixels in property "Row height". The following image shows the wider scrollbar. The size can be set under "Scrollbar size". The scrollbars are added to the table, when the table is reduced to a size that is less than the sum of the row heights or column widths.

	Order in pieces	Stocked?	Articel n ▲
0			
1			
2			

The scrollbars can be removed again by adjusting the table size. The height is calculated from the row height multiplied with the number of rows (+1 if the column header is enabled). The width is the sum of the individual column widths plus the width the label column, if this is enabled.

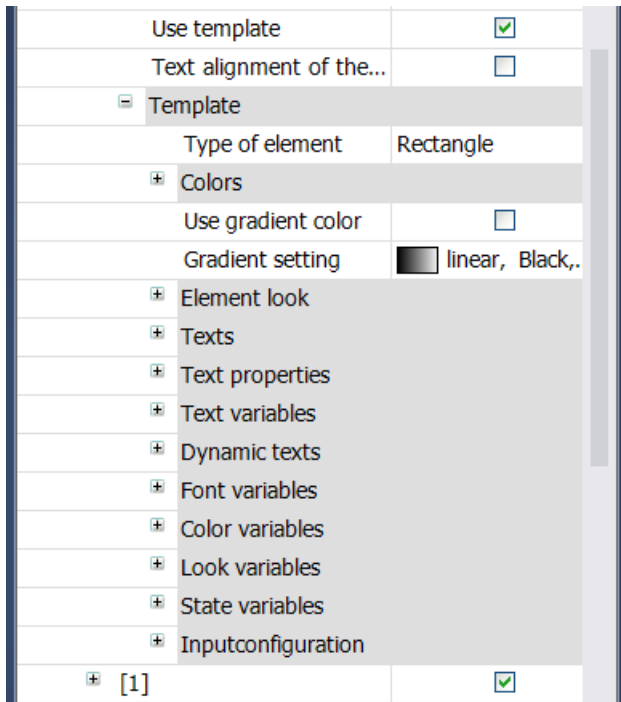
The text font and size can be selected the dialog box that opens by clicking on the  button in "Font". In the example the font color under "Font color" is changed to "Blue".

At runtime, the table looks as follows:

	Order in pieces	Stocked?	Articel number
0	250	TRUE	32136832
1	480	FALSE	45983920
2	1500	TRUE	46001235
3	680	TRUE	55129547

i Note that the horizontal alignment of the cells is specified in the settings for the respective columns under "Text alignment of the heading". Except in cases where a template is used on a per column basis, as described below.

The table fields can be edited column by column. To this end, tick the checkbox "Use template" for the selected column, whereupon "Template" is added as a further item in the column properties and expanded.



Three different template types are available for selection. "Rectangle" is entered by default. However, this can be changed to "rounded rectangle" or ellipse" by clicking inside the value field of the property "Element type". The corresponding configuration options show the element properties of the selected template. As usual, you can select the fill and frame colors for normal and alarm state, which can also be switched via a variable. The alignment of the entries in the column cells (except the header cells) is now determined by the setting in the template.

i Instead of array components entered by default, any other project variable can be entered under "Text variable". In particular, this enables the array elements to be displayed in a different order in the table.

i The columns can also be swapped within in the table. To this end, click on the center of a column within the header and drag the column to its new position while pressing the mouse button.

In this example, a template was used for the first and third column.

	Order in pieces	Stocked?	Articel number
0	250	TRUE	32136832
1	480	FALSE	45983920
2	1500	TRUE	46001235
3	680	TRUE	55129547

To make a selection for a cell visible, the settings are then adjusted under "Selection". Initially the color is set to "light blue", the selection type to "row selection". If a cell is selected at runtime, all other cells in the current row are also highlighted.

	Order in pieces	Stocked?	Articel number
0	250	TRUE	32136832
1	480	FALSE	45983920
2	1500	TRUE	46001235
3	680	TRUE	55129547

15.8.2.8 Text field

This element enables text to be displayed, which is assigned directly in the [Properties \[▶ 451\]](#) or variably in "[Text variables \[▶ 452\]](#)". In contrast to the rectangle, the frame can be displayed with a shadow.



Properties editor

The properties of a visualization element - except [alignment and order \[▶ 377\]](#) - can all be configured in the [properties editor \[▶ 385\]](#). By default, this editor opens next to the visualization editor, or it can be opened explicitly via the "Properties" command (which can be found in the View menu as standard).

A property can be modified by editing the field "Value". To this end, an input field, a selection list, a dialog or checkbox that can be activated is provided in this field, depending on the element type. The value field opens

- after a double-click,
- after a single click in a selected field,
- via the space bar, if the field was already selected.

If a variable is assigned,

- simply enter its name.
- Use the button to open the input assistant for selecting a variable. The Variables category lists all variables that have already been defined in the project.

Working in the list of properties can be made easier with the aid of default, sorting and filter functions.

Element properties

All element properties and their descriptions are listed below.


Element name	The element name can be changed. Standard name is "GenElemInst_x". "x" stands for a sequential number.
Element type	The element type is entered here. For three element groups it is possible to switch between the corresponding elements by changing the element type: <ul style="list-style-type: none"> • rectangle, rounded rectangle and ellipse [▶ 475] • Polygon, polyline or Bézier curve [▶ 490] • dip switch, power switch, push switch, push switch with LED, rocker switch [▶ 532]

Position

Here you can define the position (X/Y coordinates) and size (width and height) of the element in pixels. The origin is in the top left corner of the window. The positive x-axis is on the right, the positive y-axis runs downwards. If the values edited, the displayed element is simultaneously modified in the [visualization editor \[▶ 376\]](#).

X	Horizontal position in pixels – X=0 is the left edge of the window.
Y	Vertical position in pixels – Y=0 is the upper edge of the window.
Width	Width of the element in pixels
Height	Height of the element in pixels

Colors

A color is defined based on a hexadecimal number consisting of red / green / blue (RGB) components. For each of these three colors, 256 (0-255) values are available, which can be entered here statically. The color can be selected from a selection list or via the color selection dialog, which can be opened via the button . In addition, the level of transparency can be set for each color (0: fully transparent, 255: fully opaque).

Normal state • Frame color • Fill color	Select a frame and fill color for the default state. If the color switching variable is defined as FALSE, the element is in default state.
Alarm state • Frame color • Fill color	Select a frame and fill color for the element in alarm state. This is triggered, if the color switching variable is defined as TRUE.



Transparency is not supported under Windows CE.

Color variables

The color variables used for dynamic definition of the element colors via project variables of type DWORD. A color is defined based on a hexadecimal number consisting of red, green and blue (RGB) components. In addition, the variables are used to specify the transparency of the color (FF: fully opaque - 00: fully transparent). The DWORD has the following structure: 16#TTRRGGBB



The structure of the hexadecimal number differs, compared to TwinCAT 2. In TwinCAT 3, in addition to the RGB components, the color transparency can also be defined with the hexadecimal number. The transparency is indicated by the first two digits after "16#". Colors with definitions starting with "16#00" are invisible, since they are fully transparent.

Example:

```
nFillColor := 16#FF8FE03F;
```

- FF: transparency (fully opaque)
- 8F: red
- E0: green
- 3F: blue

Color change	Boolean variable, which controls the switching of the element color between "normal state" (variable = FALSE) and "alarm state" (variable = TRUE).
Normal state	Variables of type DWORD for defining the frame and fill color for the element. They overwrite the values that are currently defined in "Colors". The values in the project variable are used, if the variable defined in "Color change" is FALSE.
Alarm state	Variables of type DWORD for defining the frame and fill color for the element in alarm state. They overwrite the values that are currently defined in "Colors". The values in the project variable are used, if the variable defined in "Color change" is TRUE.



Transparency is not supported under Windows CE.

Appearance

The settings under Appearance are static definitions for the frame and the element filling.

Line width	Defines the frame line width in pixels. 0 codes the same as 1 and sets the line width to 1 pixel. If no frame is required, set the line type to invisible.
Fill type	Defines the fill type for the fill color: <ul style="list-style-type: none"> • Filled: The fill color is visible • Invisible: The fill color is invisible
Line type	Defines one of the following line types for the outline: <ul style="list-style-type: none"> • solid • dashed • dotted • dotdash • dash dot dot • Invisible: outline is invisible.

Auxiliary setting

Shadow style	Here you can specify the shadow style for the outline: <ul style="list-style-type: none"> • Recessed • No shadow • Raised • From style [▶ 389]: Standard setting
--------------	--



Texts

These properties are used for a static definition of the element labelling. Each can contain a [formatting sequence \[▶ 617\]](#), e.g. %s. In online mode the sequence is replaced by the contents of the variables defined in "Text variables".


Text	Enter a text. It is used to label the element. The text can contain a formatting sequence, e.g. %s. The corresponding variable is defined in "Text variables". Note: The text can also be entered directly. If the element is selected in the visualization editor, an input field can be opened by pressing the space bar.
Tooltip	Enter a text. It is used as a tooltip for the element and appears in the visualization only in online mode when the cursor is placed over an element. The text can contain a formatting sequence, e.g. %s. The corresponding variable is defined in "Text variables".

Text properties

These properties are used for a static definition of the font. A dynamic definition of the font is possible in the category "Font variables".

Horizontal alignment	Defines the horizontal alignment of the text through selection of: <ul style="list-style-type: none"> • Left • Centered • Right
Vertical alignment	Defines the vertical alignment of the text through selection of: <ul style="list-style-type: none"> • Top • Centered • Bottom
Text format	Defines the display of a text that is too long to be displayed completely in an element: <ul style="list-style-type: none"> • Default – the text extends beyond the element. • Line break – the text is automatically wrapped. • Ellipsis - the text is displayed as far as possible and then truncated with "...".
Font:	Defines the font through selection from predefined fonts: <ul style="list-style-type: none"> • Standard • Heading • Large heading • Title • Note <p>Press  to opens the dialog for user-defined font properties.</p>
Font color	Defines the font color for the element. Either from the selection list or via the dialog that opens when  is clicked.

Text variables

You can display text that is stored in a variable. To do this, first add a formatting sequence in the text that is defined under the "Texts" property. Then assign a variable. In online mode the [formatting sequence](#)  6171 is replaced by the contents of the variables.

Example %f:

Enter "Result: %2.5f" in the "Texts" property. Enter "fValue" in the "Text variables" property. The variable must be a defined IEC variable. The element will then be labeled in online mode with "Result: 12.12345" if fValue = 12.1234567.

Text variable	Variable (of default data type) containing the information to be displayed. The type must match the formatting sequence in the "Texts" setting.
Tooltipvariable	Variable of type String containing the tooltip text to be displayed. The entry in the property "Texts" must contain a formatting sequence.

State variables

These are dynamic definitions of the availability of the element in online mode.

Invisibility	Specification of a boolean variable. If this returns TRUE, the element is invisible in online mode.
Input disabled	Specification of a boolean variable. If TRUE is returned, inputs for the element have no effect. Also, the element itself is greyed out in the visualization, to indicate that no user inputs are possible. If the visualization uses the user management [▶ 393], the elements for user groups with access right "only visible" are greyed out.

Input configuration

Here you can define the consequential action that should be performed when the user makes an input in the element in online mode. As long as no follow-on actions are defined, "Configure..." appears in the Properties field. Click on "Configure..." to open the [Input Configuration](#) [[▶ 406](#)], where you can assign follow-on actions. Each input action can be assigned any number of follow-on actions.

The following input events are available for an element:

- OnDialogClosed
- OnMouseClicked
- OnMouseDown
- OnMouseEnter
- OnMouseLeave
- OnMouseMove
- OnMouseUp

OnDialogClosed	This event is triggered if one of the dialog boxes that were opened for user inputs is closed within a visualization. Note: This property is not limited to the element, for which it is configured, but applies within the entire visualization. It therefore does not respond to each dialog closing action. Currently there is no way to define such a property for the entire visualization. It therefore has to be assigned to one of its elements.
OnMouseClicked	This mouse event is triggered when the cursor points to an element and a full mouse click (pressing and releasing the mouse button) is executed on this item.
OnMouseDown	This mouse event is triggered when the mouse button is pressed while the cursor points to an element. It is irrelevant where on the visualization the mouse button is released again.
OnMouseEnter	This mouse event is triggered when the cursor is dragged to the element. It is irrelevant whether the mouse button is pressed or released.
OnMouseLeave	This mouse event is triggered when the cursor leaves the element. It is irrelevant whether the mouse button is pressed or released.
OnMouseMove	This mouse event is triggered when the cursor is moved within the element. It is irrelevant whether the mouse button is pressed or released.
OnMouseUp	This mouse event occurs when the mouse button is released on the element. The mouse button was pressed prior to that outside of the element.

Input configuration - hotkeys

A hotkey can be used to define a key or key combination and link it with a follow-on action (e.g. MouseDown, MouseUp), which is to be executed when a key event occurs (KeyDown, KeyUp). By default the MouseDown action is executed on KeyDown (press key) and the MouseUp action on KeyUp (release key). This can be useful if a visualization is to be operated both via mouse actions and keyboard entries, since the input actions then only have to be configured once. The key configuration for an element is also managed in the [hotkeys configuration](#) [[▶ 392](#)] for the visualization. Any changes are always synchronized between this and the element properties editor.

Key	Assigning a key. A selection list contains all currently supported keys, e.g. M.
Event(s)	Definition of the event to be executed, if the key or key and modifier are used. Possible values that are available in a selection list: <ul style="list-style-type: none"> • No action • MouseDown action when the key is pressed • MouseUp action when the key is released • MouseDown/MouseUp action when the key is pressed/released
Shift	If this option is enabled, the key has to be used in combination with the Shift key.
Control	If this option is enabled, the key has to be used in combination with the Ctrl key.
Alt	If this option is enabled, the key has to be used in combination with the Alt key.

Input configuration – Keys

"Keys" can be used to specify that the value of a boolean project variable is set depending on the mouse behavior for the event "Keys".

Variable	Boolean variable whose value is TRUE if the mouse button is pressed while the cursor points to the element. The value becomes FALSE again when the mouse button is released or the cursor leaves the element.
FALSE keys	If this option is enabled, the key behavior described above is reversed for the corresponding variable. That is, when the mouse button pressed, the variable value is set to FALSE. When the button is, the variable value is set to TRUE.
Switch on enter if mouse trapped	If this option is enabled, the variable value behaves as described under 'Variables', as long as the cursor is within the element area. If the cursor leaves the element area while the mouse button is pressed, the variable is set to FALSE. However, it is automatically set to TRUE again, if the cursor returns to the element area without the mouse button being released. In other words, the system takes into account that the mouse is "trapped" as long as the mouse button is pressed, even if the cursor leaves the element area.

Input configuration - Switching

"Switching" can be used to specify that for this event the value of a boolean project variable is set depending on the mouse behavior.

Variable	Boolean variable whose value switches between TRUE and FALSE with each mouse click on the element. No switching occurs if the cursor is moved away from the element while the mouse button is pressed. Can be used to cancel a switching action.
Switch on release, if mouse trapped	If this option is enabled, the variable value behaves as described under 'Variables', as long as the cursor is within the element area. If the cursor leaves the element area while the mouse button is pressed and the mouse button is then released, the variable is switched anyway. In other words, the system takes into account that the mouse is "trapped" as long as the mouse button is pressed, even if the cursor leaves the element area.

Access rights

This setting relates to the access rights for the individual element. Click to open the [Access rights dialog \[▶ 420\]](#). The setting is only available if a [user management \[▶ 393\]](#) was added to the PLC project. The following status messages are available:

Not set. All rights.	The default message is set, if the element is shown as available for all groups.
Rights are issued: Limited rights.	The message is set, if the element is shown with limited behavior for at least one group.

15.8.2.9 Scrollbar

This element generates a scrollbar. If a user drags the scroll box to a different position while the visualization is running, the property "[Value \[▶ 455\]](#)" is set accordingly. The alignment of the scrollbar can be [changed \[▶ 456\]](#) from horizontal to vertical.



Properties editor

The properties of a visualization element - except alignment and order [[▶ 377](#)] - can all be configured in the properties editor [[▶ 385](#)]. By default, this editor opens next to the visualization editor, or it can be opened explicitly via the "Properties" command (which can be found in the View menu as standard).

A property can be modified by editing the field "Value". To this end, an input field, a selection list, a dialog or checkbox that can be activated is provided in this field, depending on the element type. The value field opens

- after a double-click,
- after a single click in a selected field,
- via the space bar, if the field was already selected.

If a variable is assigned,

- simply enter its name.
- Use the button to open the input assistant for selecting a variable. The Variables category lists all variables that have already been defined in the project.

Working in the list of properties can be made easier with the aid of default, sorting and filter functions.

Element properties

All element properties and their descriptions are listed below.

Element name	The element name can be changed. Standard name is "GenElemInst_x". "x" stands for a sequential number.
Element type	The element type is entered here. For three element groups it is possible to switch between the corresponding elements by changing the element type: <ul style="list-style-type: none"> • rectangle, rounded rectangle and ellipse [▶ 475] • Polygon, polyline or Bézier curve [▶ 490] • dip switch, power switch, push switch, push switch with LED, rocker switch [▶ 532]

Additional settings

Value	Variable of type integer, which contains the position of the scroll box in the scrollbar.
Minimum value	Smallest possible scrollbar value
Maximum value	Largest possible scrollbar value
Page size	If a user clicks in the scrollbar at runtime, the position of the scroll box is moved by the page size in the direction of the mouse position. The page size can be specified as a fixed value in pixels or as a variable of type integer.

Position

Here you can define the position (X/Y coordinates) and size (width and height) of the element in pixels. The origin is in the top left corner of the window. The positive x-axis is on the right, the positive y-axis runs downwards. If the values edited, the displayed element is simultaneously modified in the visualization editor [[▶ 376](#)].

X	Horizontal position in pixels – X=0 is the left edge of the window.
Y	Vertical position in pixels – Y=0 is the upper edge of the window.
Width	Width of the element in pixels
Height	Height of the element in pixels

Bar

Alignment	<p>Here you can specify the alignment of the bar:</p> <ul style="list-style-type: none"> • Horizontal • Vertical <p>The alignment can also be configured directly in the visualization, by changing the width and height of the scrollbar.</p>
Direction of travel	<p>If the alignment is horizontal, you can select the scrolling direction as follows:</p> <ul style="list-style-type: none"> • Left to right • Right to left <p>If the alignment is vertical, you can select the scrolling direction as follows:</p> <ul style="list-style-type: none"> • From bottom to top • From top to bottom

Colors

A color is defined based on a hexadecimal number consisting of red / green / blue (RGB) components. For each of these three colors, 256 (0-255) values are available, which can be entered here statically. The color can be selected from a selection list or via the color selection dialog, which can be opened via the button



. In addition, the level of transparency can be set for each color (0: fully transparent, 255: fully opaque).

<p>Normal state</p> <ul style="list-style-type: none"> • Frame color • Fill color 	<p>Select a frame and fill color for the default state. If the color switching variable is defined as FALSE, the element is in default state.</p>
<p>Alarm state</p> <ul style="list-style-type: none"> • Frame color • Fill color 	<p>Select a frame and fill color for the element in alarm state. This is triggered, if the color switching variable is defined as TRUE.</p>



Transparency is not supported under Windows CE.

Color variables

The color variables used for dynamic definition of the element colors via project variables of type DWORD. A color is defined based on a hexadecimal number consisting of red, green and blue (RGB) components. In addition, the variables are used to specify the transparency of the color (FF: fully opaque - 00: fully transparent). The DWORD has the following structure: 16#TTRRGGBB



The structure of the hexadecimal number differs, compared to TwinCAT 2. In TwinCAT 3, in addition to the RGB components, the color transparency can also be defined with the hexadecimal number. The transparency is indicated by the first two digits after "16#". Colors with definitions starting with "16#00" are invisible, since they are fully transparent.

Example:

nFillColor := 16#FF8FE03F;

- FF: transparency (fully opaque)
- 8F: red
- E0: green
- 3F: blue

Color change	Boolean variable, which controls the switching of the element color between "normal state" (variable = FALSE) and "alarm state" (variable = TRUE).
Normal state • Frame color • Fill color	Variables of type DWORD for defining the frame and fill color for the element. They overwrite the values that are currently defined in "Colors". The values in the project variable are used, if the variable defined in "Color change" is FALSE.
Alarm state • Frame color • Fill color	Variables of type DWORD for defining the frame and fill color for the element in alarm state. They overwrite the values that are currently defined in "Colors". The values in the project variable are used, if the variable defined in "Color change" is TRUE.



Transparency is not supported under Windows CE.



Texts

These properties are used for a static definition of the element labelling. Each can contain a formatting sequence [▶ 617], e.g. %s. In online mode the sequence is replaced by the contents of the variables defined in "Text variables".

Text	Enter a text. It is used to label the element. The text can contain a formatting sequence, e.g. %s. The corresponding variable is defined in "Text variables". Note: The text can also be entered directly. If the element is selected in the visualization editor, an input field can be opened by pressing the space bar.
Tooltip	Enter a text. It is used as a tooltip for the element and appears in the visualization only in online mode when the cursor is placed over an element. The text can contain a formatting sequence, e.g. %s. The corresponding variable is defined in "Text variables".

Text properties

These properties are used for a static definition of the font. A dynamic definition of the font is possible in the category "Font variables".

Horizontal alignment	Defines the horizontal alignment of the text through selection of: <ul style="list-style-type: none"> • Left • Centered • Right
Vertical alignment	Defines the vertical alignment of the text through selection of: <ul style="list-style-type: none"> • Top • Centered • Bottom
Text format	Defines the display of a text that is too long to be displayed completely in an element: <ul style="list-style-type: none"> • Default – the text extends beyond the element. • Line break – the text is automatically wrapped. • Ellipsis - the text is displayed as far as possible and then truncated with "...".
Font:	Defines the font through selection from predefined fonts: <ul style="list-style-type: none"> • Standard • Heading • Large heading • Title • Note Press  to opens the dialog for user-defined font properties.
Font color	Defines the font color for the element. Either from the selection list or via the dialog that opens when  is clicked.

Text variables

You can display text that is stored in a variable. To do this, first add a formatting sequence in the text that is defined under the "Texts" property. Then assign a variable. In online mode the [formatting sequence \[▶ 617\]](#) is replaced by the contents of the variables.

Example %f:

Enter "Result: %2.5f" in the "Texts" property. Enter "fValue" in the "Text variables" property. The variable must be a defined IEC variable. The element will then be labeled in online mode with "Result: 12.12345" if fValue = 12.1234567.

Text variable	Variable (of default data type) containing the information to be displayed. The type must match the formatting sequence in the "Texts" setting.
Tooltiptvariable	Variable of type String containing the tooltip text to be displayed. The entry in the property "Texts" must contain a formatting sequence.

Dynamic texts

These parameters are used to define dynamic texts originating from [text lists \[▶ 138\]](#). This enables [language change \[▶ 616\]](#), for example.

A further possibility to define a text dynamically is to supply the text via a string variable. (see the category "Text variables")

Text list	Name of the text list, as used in the project tree, as a string Example: 'TL_ErrorList'
Textindex	Index (ID) of the text, as defined in the text list, as a string. It can be specified directly in a static manner or as a string variable.
Tooltiptindex	Index (ID) of the tooltip text, as defined in the text list, as a string. It can be specified directly in a static manner or as a string variable.

Font variables

These variables are used for dynamic font definitions for element texts via project variables. Static definitions are configured under "Text properties".

Font name	Specification of a variable of the type String that contains the font name that is to be used to label the element. (name specified as in the standard font dialog) Example: MAIN.sFont (sFont := 'Arial');
Size	Variable of type INT containing the size of the element text in pixels, as in the default dialog 'Font'. Example: MAIN.nHeight (nHeight := 16;)
Flags	Variable of type DWORD for defining the font display via one of the flag values listed below. A combined definition can be achieved by adding the respective flag values and specifying the sum. <ul style="list-style-type: none"> • 1: italics • 2: bold • 4: underlined • 8: deleted Example: MAIN.nFlag (nFlag := 6;) The text is shown bold and underlined.
Character set	The character set to be used for the font can be defined via the default font number. This number can be specified via a DWORD variable (see also the definition in the default font dialog)
Color	Variable of type DWORD for defining the color of the element text.
Flags for the text alignment	Variable of type DWORD for defining the text alignment via one of the flag values listed below. A combined definition can be achieved by adding the respective flag values and specifying the sum. <ul style="list-style-type: none"> • 0: top left • 1: centered horizontally • 2: right • 4: centered vertically • 8: below Example MAIN.nFlag (nFlag := 5;) The text is displayed centered horizontally and vertically.

Access rights

This setting relates to the access rights for the individual element. Click to open the [Access rights dialog](#) [► 420]. The setting is only available if a [user management](#) [► 393] was added to the PLC project. The following status messages are available:

Not set. All rights.	The default message is set, if the element is shown as available for all groups.
Rights are issued: Limited rights.	The message is set, if the element is shown with limited behavior for at least one group.

State variables

These are dynamic definitions of the availability of the element in online mode.

Invisibility	Specification of a boolean variable. If this returns TRUE, the element is invisible in online mode.
Input disabled	Specification of a boolean variable. If TRUE is returned, inputs for the element have no effect. Also, the element itself is greyed out in the visualization, to indicate that no user inputs are possible. If the visualization uses the user management [► 393] , the elements for user groups with access right "only visible" are greyed out.

15.8.2.10 Slider

The slider bar can be moved with the mouse, whereby a variable assigned to the slider changes its value within the defined limits.



Properties editor

The properties of a visualization element - except [alignment and order \[► 377\]](#) - can all be configured in the [properties editor \[► 385\]](#). By default, this editor opens next to the visualization editor, or it can be opened explicitly via the "Properties" command (which can be found in the View menu as standard).

A property can be modified by editing the field "Value". To this end, an input field, a selection list, a dialog or checkbox that can be activated is provided in this field, depending on the element type. The value field opens

- after a double-click,
- after a single click in a selected field,
- via the space bar, if the field was already selected.

If a variable is assigned,

- simply enter its name.
- Use the button to open the input assistant for selecting a variable. The Variables category lists all variables that have already been defined in the project.

Working in the list of properties can be made easier with the aid of default, sorting and filter functions.

Element properties

All element properties and their descriptions are listed below.

Element name	The element name can be changed. Standard name is "GenElemInst_x". "x" stands for a sequential number.
Element type	The element type is entered here. For three element groups it is possible to switch between the corresponding elements by changing the element type: <ul style="list-style-type: none"> • rectangle, rounded rectangle and ellipse [► 475] • Polygon, polyline or Bézier curve [► 490] • dip switch, power switch, push switch, push switch with LED, rocker switch [► 532]

Position

Here you can define the position (X/Y coordinates) and size (width and height) of the element in pixels. The origin is in the top left corner of the window. The positive x-axis is on the right, the positive y-axis runs downwards. If the values edited, the displayed element is simultaneously modified in the [visualization editor \[► 376\]](#).

X	Horizontal position in pixels – X=0 is the left edge of the window.
Y	Vertical position in pixels – Y=0 is the upper edge of the window.
Width	Width of the element in pixels
Height	Height of the element in pixels

Auxiliary setting

Variable	Numeric variable, which will contain the position of the slider.
-----------------	--

Scale

Show scale	The scale is shown if this option is enabled.
Scale start	Lower scale limit value
Scale end	Upper scale limit value
Main scale	Distance between two lines of the coarse scale
Sub scale	Distance between two lines on the fine scale. The value can be set to 0 if a further subdivision of the coarse scale is not desired.
Scale format (C-Syntax)	If a scale format, such as “%d s” is configured, the scale is displayed with labels.
Scale ratio	Size of the scale in % of the total size

Bars

Chart type	The drop-down list offers various options for positioning bars and scales: <ul style="list-style-type: none"> • Scale beside bar • Scale in bar • Bar in scale • No scale
Alignment	The bar can be aligned horizontally or vertically. The alignment is derived from the ratio between the width and height and cannot be edited here. It can be changed in the visualization editor [▶ 376] by "gripping" a corner point of the element with the mouse and dragging it horizontally or vertically.
Direction of movement	If the alignment is horizontal, there is a choice between: <ul style="list-style-type: none"> • Left to right • Right to left If the alignment is vertical, there is a choice between: <ul style="list-style-type: none"> • From bottom to top • From top to bottom

State variables

These are dynamic definitions of the availability of the element in online mode.

Invisibility	Specification of a boolean variable. If this returns TRUE, the element is invisible in online mode.
Input disabled	Specification of a boolean variable. If TRUE is returned, inputs for the element have no effect. Also, the element itself is greyed out in the visualization, to indicate that no user inputs are possible. If the visualization uses the user management [▶ 393] , the elements for user groups with access right "only visible" are greyed out.

Access rights

This setting relates to the access rights for the individual element. Click to open the [Access rights dialog \[► 420\]](#). The setting is only available if a [user management \[► 393\]](#) was added to the PLC project. The following status messages are available:

Not set. All rights.	The default message is set, if the element is shown as available for all groups.
Rights are issued: Limited rights.	The message is set, if the element is shown with limited behavior for at least one group.

15.8.2.11 SpinControl

The element "SpinControl" enables the value of a variable to be incremented or decremented by clicking on the small arrow keys to the right of the element.




Properties editor

The properties of a visualization element - except [alignment and order \[► 377\]](#) - can all be configured in the [properties editor \[► 385\]](#). By default, this editor opens next to the visualization editor, or it can be opened explicitly via the "Properties" command (which can be found in the View menu as standard).

A property can be modified by editing the field "Value". To this end, an input field, a selection list, a dialog or checkbox that can be activated is provided in this field, depending on the element type. The value field opens

- after a double-click,
- after a single click in a selected field,
- via the space bar, if the field was already selected.

If a variable is assigned,

- simply enter its name.
- Use the  button to open the input assistant for selecting a variable. The Variables category lists all variables that have already been defined in the project.

Working in the list of properties can be made easier with the aid of default, sorting and filter functions.

Element properties

All element properties and their descriptions are listed below.

Element name	The element name can be changed. Standard name is "GenElemInst_x". "x" stands for a sequential number.
Element type	The element type is entered here. For three element groups it is possible to switch between the corresponding elements by changing the element type: <ul style="list-style-type: none"> • rectangle, rounded rectangle and ellipse [► 475] • Polygon, polyline or Bézier curve [► 490] • dip switch, power switch, push switch, push switch with LED, rocker switch [► 532]

Position

Here you can define the position (X/Y coordinates) and size (width and height) of the element in pixels. The origin is in the top left corner of the window. The positive x-axis is on the right, the positive y-axis runs downwards. If the values edited, the displayed element is simultaneously modified in the [visualization editor \[► 376\]](#).

X	Horizontal position in pixels – X=0 is the left edge of the window.
Y	Vertical position in pixels – Y=0 is the upper edge of the window.
Width	Width of the element in pixels
Height	Height of the element in pixels

Additional settings

Variable	Numerical variable, the value of which increases or decreases after a user input.
Number format	Formatting sequence, with which the variable value is displayed. Example: %i for an integer variable
Interval	Interval (increment) by which the value of the variable is incremented or decremented after a user input.



Value range

Minimum value	Enter the maximum value.
Maximum value	Enter the minimum value.

Text properties

Use of	The following two settings are available for selection: <ul style="list-style-type: none"> • Default style values • User-defined settings <ul style="list-style-type: none"> ◦ Horizontal alignment ◦ Vertical alignment ◦ Font ◦ Font color
--------	---

User-defined text properties

Horizontal alignment	Defines the horizontal alignment of the text through selection of: <ul style="list-style-type: none"> • Left • Centered • Right
Vertical alignment	Defines the vertical alignment of the text through selection of: <ul style="list-style-type: none"> • Top • Centered • Bottom
Font:	Defines the font through selection from predefined fonts: <ul style="list-style-type: none"> • Standard • Heading • Large heading • Title • Note <p>Press  to opens the dialog for user-defined font properties.</p>
Font color	Defines the font color for the element. Either from the selection list or via the dialog that opens when  is clicked.

Color variables

The color variables used for dynamic definition of the element colors via project variables of type DWORD. A color is defined based on a hexadecimal number consisting of red, green and blue (RGB) components. In addition, the variables are used to specify the transparency of the color (FF: fully opaque - 00: fully transparent). The DWORD has the following structure: 16#TTRRGGBB

i The structure of the hexadecimal number differs, compared to TwinCAT 2. In TwinCAT 3, in addition to the RGB components, the color transparency can also be defined with the hexadecimal number. The transparency is indicated by the first two digits after "16#". Colors with definitions starting with "16#00" are invisible, since they are fully transparent.

Example:

nFillColor := 16#FF8FE03F;

- FF: transparency (fully opaque)
- 8F: red
- E0: green
- 3F: blue

Color change	Boolean variable, which controls the switching of the element color between "normal state" (variable = FALSE) and "alarm state" (variable = TRUE).
--------------	--

i Transparency is not supported under Windows CE.

State variables

These are dynamic definitions of the availability of the element in online mode.

Invisibility	Specification of a boolean variable. If this returns TRUE, the element is invisible in online mode.
Input disabled	Specification of a boolean variable. If TRUE is returned, inputs for the element have no effect. Also, the element itself is greyed out in the visualization, to indicate that no user inputs are possible. If the visualization uses the <u>user management</u> [▶ 393], the elements for user groups with access right "only visible" are greyed out.

Input configuration

Here you can define the consequential action that should be performed when the user makes an input in the element in online mode. As long as no follow-on actions are defined, "Configure..." appears in the Properties field. Click on "Configure..." to open the Input Configuration [▶ 406], where you can assign follow-on actions. Each input action can be assigned any number of follow-on actions.

The following input events are available for an element:

- OnDialogClosed
- OnMouseClicked
- OnMouseDown
- OnMouseEnter
- OnMouseLeave
- OnMouseMove
- OnMouseUp

OnDialogClosed	This event is triggered if one of the dialog boxes that were opened for user inputs is closed within a visualization. Note: This property is not limited to the element, for which it is configured, but applies within the entire visualization. It therefore does not respond to each dialog closing action. Currently there is no way to define such a property for the entire visualization. It therefore has to be assigned to one of its elements.
OnMouseClicked	This mouse event is triggered when the cursor points to an element and a full mouse click (pressing and releasing the mouse button) is executed on this item.
OnMouseDown	This mouse event is triggered when the mouse button is pressed while the cursor points to an element. It is irrelevant where on the visualization the mouse button is released again.
OnMouseEnter	This mouse event is triggered when the cursor is dragged to the element. It is irrelevant whether the mouse button is pressed or released.
OnMouseLeave	This mouse event is triggered when the cursor leaves the element. It is irrelevant whether the mouse button is pressed or released.
OnMouseMove	This mouse event is triggered when the cursor is moved within the element. It is irrelevant whether the mouse button is pressed or released.
OnMouseUp	This mouse event occurs when the mouse button is released on the element. The mouse button was pressed prior to that outside of the element.

Input configuration - hotkeys

A hotkey can be used to define a key or key combination and link it with a follow-on action (e.g. MouseDown, MouseUp), which is to be executed when a key event occurs (KeyDown, KeyUp). By default the MouseDown action is executed on KeyDown (press key) and the MouseUp action on KeyUp (release key). This can be useful if a visualization is to be operated both via mouse actions and keyboard entries, since the input actions then only have to be configured once. The key configuration for an element is also managed in the [hotkeys configuration \[▶ 392\]](#) for the visualization. Any changes are always synchronized between this and the element properties editor.

Key	Assigning a key. A selection list contains all currently supported keys, e.g. M.
Event(s)	Definition of the event to be executed, if the key or key and modifier are used. Possible values that are available in a selection list: <ul style="list-style-type: none"> • No action • MouseDown action when the key is pressed • MouseUp action when the key is released • MouseDown/MouseUp action when the key is pressed/released
Shift	If this option is enabled, the key has to be used in combination with the Shift key.
Control	If this option is enabled, the key has to be used in combination with the Ctrl key.
Alt	If this option is enabled, the key has to be used in combination with the Alt key.

Access rights

This setting relates to the access rights for the individual element. Click to open the [Access rights dialog \[▶ 420\]](#). The setting is only available if a [user management \[▶ 393\]](#) was added to the PLC project. The following status messages are available:

Not set. All rights.	The default message is set, if the element is shown as available for all groups.
Rights are issued: Limited rights.	The message is set, if the element is shown with limited behavior for at least one group.

15.8.2.12 Invisible input

The element is displayed in the editor with a dashed line, but this is invisible in online mode. The behavior of the element has to be specified in the input configuration.




Properties editor

The properties of a visualization element - except alignment and order [[▶ 377](#)] - can all be configured in the properties editor [[▶ 385](#)]. By default, this editor opens next to the visualization editor, or it can be opened explicitly via the "Properties" command (which can be found in the View menu as standard).

A property can be modified by editing the field "Value". To this end, an input field, a selection list, a dialog or checkbox that can be activated is provided in this field, depending on the element type. The value field opens

- after a double-click,
- after a single click in a selected field,
- via the space bar, if the field was already selected.

If a variable is assigned,

- simply enter its name.
- Use the  button to open the input assistant for selecting a variable. The Variables category lists all variables that have already been defined in the project.

Working in the list of properties can be made easier with the aid of default, sorting and filter functions.

Element properties

All element properties and their descriptions are listed below.

Element name	The element name can be changed. Standard name is "GenElemInst_x". "x" stands for a sequential number.
Element type	The element type is entered here. For three element groups it is possible to switch between the corresponding elements by changing the element type: <ul style="list-style-type: none"> • rectangle, rounded rectangle and ellipse [▶ 475] • Polygon, polyline or Bézier curve [▶ 490] • dip switch, power switch, push switch, push switch with LED, rocker switch [▶ 532]

Position

Here you can define the position (X/Y coordinates) and size (width and height) of the element in pixels. The origin is in the top left corner of the window. The positive x-axis is on the right, the positive y-axis runs downwards. If the values edited, the displayed element is simultaneously modified in the visualization editor [[▶ 376](#)].

X	Horizontal position in pixels – X=0 is the left edge of the window.
Y	Vertical position in pixels – Y=0 is the upper edge of the window.
Width	Width of the element in pixels
Height	Height of the element in pixels

State variables

These are dynamic definitions of the availability of the element in online mode.

Invisibility	Specification of a boolean variable. If this returns TRUE, the element is invisible in online mode.
Input disabled	Specification of a boolean variable. If TRUE is returned, inputs for the element have no effect. Also, the element itself is greyed out in the visualization, to indicate that no user inputs are possible. If the visualization uses the user management [▶ 393] , the elements for user groups with access right "only visible" are grayed out.

Input configuration

Here you can define the consequential action that should be performed when the user makes an input in the element in online mode. As long as no follow-on actions are defined, "Configure..." appears in the Properties field. Click on "Configure..." to open the [Input Configuration \[▶ 406\]](#), where you can assign follow-on actions. Each input action can be assigned any number of follow-on actions.

The following input events are available for an element:

- OnDialogClosed
- OnMouseClicked
- OnMouseDown
- OnMouseEnter
- OnMouseLeave
- OnMouseMove
- OnMouseUp

OnDialogClosed	This event is triggered if one of the dialog boxes that were opened for user inputs is closed within a visualization. Note: This property is not limited to the element, for which it is configured, but applies within the entire visualization. It therefore does not respond to each dialog closing action. Currently there is no way to define such a property for the entire visualization. It therefore has to be assigned to one of its elements.
OnMouseClicked	This mouse event is triggered when the cursor points to an element and a full mouse click (pressing and releasing the mouse button) is executed on this item.
OnMouseDown	This mouse event is triggered when the mouse button is pressed while the cursor points to an element. It is irrelevant where on the visualization the mouse button is released again.
OnMouseEnter	This mouse event is triggered when the cursor is dragged to the element. It is irrelevant whether the mouse button is pressed or released.
OnMouseLeave	This mouse event is triggered when the cursor leaves the element. It is irrelevant whether the mouse button is pressed or released.
OnMouseMove	This mouse event is triggered when the cursor is moved within the element. It is irrelevant whether the mouse button is pressed or released.
OnMouseUp	This mouse event occurs when the mouse button is released on the element. The mouse button was pressed prior to that outside of the element.

Input configuration - hotkeys

A hotkey can be used to define a key or key combination and link it with a follow-on action (e.g. MouseDown, MouseUp), which is to be executed when a key event occurs (KeyDown, KeyUp). By default the MouseDown action is executed on KeyDown (press key) and the MouseUp action on KeyUp (release key). This can be useful if a visualization is to be operated both via mouse actions and keyboard entries, since the input actions then only have to be configured once. The key configuration for an element is also managed in the [hotkeys configuration \[▶ 392\]](#) for the visualization. Any changes are always synchronized between this and the element properties editor.

Key	Assigning a key. A selection list contains all currently supported keys, e.g. M.
Event(s)	Definition of the event to be executed, if the key or key and modifier are used. Possible values that are available in a selection list: <ul style="list-style-type: none"> • No action • MouseDown action when the key is pressed • MouseUp action when the key is released • MouseDown/MouseUp action when the key is pressed/released
Shift	If this option is enabled, the key has to be used in combination with the Shift key.
Control	If this option is enabled, the key has to be used in combination with the Ctrl key.
Alt	If this option is enabled, the key has to be used in combination with the Alt key.

State variables

These are dynamic definitions of the availability of the element in online mode.

Input disabled	Boolean variable. If TRUE is returned, inputs for the element have no effect. Also, the element itself is greyed out in the visualization, to indicate that no user inputs are possible. If the visualization uses the user management [▶ 393] , the elements for user groups with access right "only visible" are greyed out.
----------------	---

Access rights

This setting relates to the access rights for the individual element. Click to open the [Access rights dialog \[▶ 420\]](#). The setting is only available if a [user management \[▶ 393\]](#) was added to the PLC project. The following status messages are available:

Not set. All rights.	The default message is set, if the element is shown as available for all groups.
Rights are issued: Limited rights.	The message is set, if the element is shown with limited behavior for at least one group.

15.8.2.13 Progress bar

A progress bar illustrates the progress of an operation. This requires a variable with a value for display in form of a bar. The [limits \[▶ 469\]](#) for the variable and the [display style \[▶ 469\]](#) can be set.




Properties editor

The properties of a visualization element - except [alignment and order \[▶ 377\]](#) - can all be configured in the [properties editor \[▶ 385\]](#). By default, this editor opens next to the visualization editor, or it can be opened explicitly via the "Properties" command (which can be found in the View menu as standard).

A property can be modified by editing the field "Value". To this end, an input field, a selection list, a dialog or checkbox that can be activated is provided in this field, depending on the element type. The value field opens

- after a double-click,
- after a single click in a selected field,
- via the space bar, if the field was already selected.

If a variable is assigned,

- simply enter its name.
- Use the  button to open the input assistant for selecting a variable. The Variables category lists all variables that have already been defined in the project.

Working in the list of properties can be made easier with the aid of default, sorting and filter functions.

Element properties

All element properties and their descriptions are listed below.

Element name	The element name can be changed. Standard name is "GenElemInst_x". "x" stands for a sequential number.
Element type	The element type is entered here. For three element groups it is possible to switch between the corresponding elements by changing the element type: <ul style="list-style-type: none"> • rectangle, rounded rectangle and ellipse [► 475] • Polygon, polyline or Bézier curve [► 490] • dip switch, power switch, push switch, push switch with LED, rocker switch [► 532]

Additional settings

Variable	Numeric variable, the value of which is visualized as a progress bar.
Minimum value	Minimum value
Maximum value	Maximum value
Style	Select one of the following styles: <ul style="list-style-type: none"> • Blocks • Bars

Position

Here you can define the position (X/Y coordinates) and size (width and height) of the element in pixels. The origin is in the top left corner of the window. The positive x-axis is on the right, the positive y-axis runs downwards. If the values edited, the displayed element is simultaneously modified in the [visualization editor \[► 376\]](#).

X	Horizontal position in pixels – X=0 is the left edge of the window.
Y	Vertical position in pixels – Y=0 is the upper edge of the window.
Width	Width of the element in pixels
Height	Height of the element in pixels

Texts

Text	Here you can specify the text to be output to the left and above the button bar. If the element is aligned horizontally, the text is positioned on the left. If the element is aligned vertically, the text is positioned at the top.
------	---

State variables

These are dynamic definitions of the availability of the element in online mode.

Invisibility	Boolean variable. If this returns TRUE, the element is invisible in online mode.
--------------	--

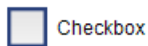
Access rights

This setting relates to the access rights for the individual element. Click to open the [Access rights dialog \[► 420\]](#). The setting is only available if a [user management \[► 393\]](#) was added to the PLC project. The following status messages are available:

Not set. All rights.	The default message is set, if the element is shown as available for all groups.
Rights are issued: Limited rights.	The message is set, if the element is shown with limited behavior for at least one group.

15.8.2.14 Checkbox

A checkbox can be used to set and reset a boolean variable. If the box is ticked, the variable is set to TRUE.




Properties editor

The properties of a visualization element - except alignment and order [[▶ 377](#)] - can all be configured in the properties editor [[▶ 385](#)]. By default, this editor opens next to the visualization editor, or it can be opened explicitly via the "Properties" command (which can be found in the View menu as standard).

A property can be modified by editing the field "Value". To this end, an input field, a selection list, a dialog or checkbox that can be activated is provided in this field, depending on the element type. The value field opens

- after a double-click,
- after a single click in a selected field,
- via the space bar, if the field was already selected.

If a variable is assigned,

- simply enter its name.
- Use the  button to open the input assistant for selecting a variable. The Variables category lists all variables that have already been defined in the project.

Working in the list of properties can be made easier with the aid of default, sorting and filter functions.

Element properties

All element properties and their descriptions are listed below.

Element name	The element name can be changed. Standard name is "GenElemInst_x". "x" stands for a sequential number.
Element type	The element type is entered here. For three element groups it is possible to switch between the corresponding elements by changing the element type: <ul style="list-style-type: none"> • <u>rectangle, rounded rectangle and ellipse</u> [▶ 475] • <u>Polygon, polyline or Bézier curve</u> [▶ 490] • <u>dip switch, power switch, push switch, push switch with LED, rocker switch</u> [▶ 532]

Position

Here you can define the position (X/Y coordinates) and size (width and height) of the element in pixels. The origin is in the top left corner of the window. The positive x-axis is on the right, the positive y-axis runs downwards. If the values edited, the displayed element is simultaneously modified in the visualization editor [[▶ 376](#)].

X	Horizontal position in pixels – X=0 is the left edge of the window.
Y	Vertical position in pixels – Y=0 is the upper edge of the window.
Width	Width of the element in pixels
Height	Height of the element in pixels

Additional settings

Variable	Boolean variable, in which the state of the checkbox is stored.
Frame size	Frame size of the checkbox Standard setting: From <u>style</u> [▶ 389]

Texts

These properties are used for a static definition of the element labelling.

Text	Enter a text. It is used to label the element. Note: The text can also be entered directly. If the element is selected in the visualization editor, an input field can be opened by pressing the space bar.
Tooltip	Enter a text. It is used as a tooltip for the element and appears in the visualization only in online mode when the cursor is placed over an element.

Text properties

Use of	The following two settings are available for selection: <ul style="list-style-type: none"> • Default style values • User-defined settings <ul style="list-style-type: none"> ◦ Horizontal alignment ◦ Vertical alignment ◦ Font ◦ Font color
--------	---

User-defined text properties

Horizontal alignment	Defines the horizontal alignment of the text through selection of: <ul style="list-style-type: none"> • Left • Centered • Right
Vertical alignment	Defines the vertical alignment of the text through selection of: <ul style="list-style-type: none"> • Top • Centered • Bottom
Font:	Defines the font through selection from predefined fonts: <ul style="list-style-type: none"> • Standard • Heading • Large heading • Title • Note <p>Press <input type="button" value="..."/> to opens the dialog for user-defined font properties.</p>
Font color	Defines the font color for the element. Either from the selection list or via the dialog that opens when <input type="button" value="..."/> is clicked.

State variables

These are dynamic definitions of the availability of the element in online mode.


Invisibility	Specification of a boolean variable. If this returns TRUE, the element is invisible in online mode.
Input disabled	Specification of a boolean variable. If TRUE is returned, inputs for the element have no effect. Also, the element itself is greyed out in the visualization, to indicate that no user inputs are possible. If the visualization uses the <u>user management</u> [▶ 393], the elements for user groups with access right "only visible" are grayed out.

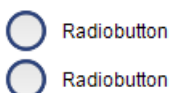
Access rights

This setting relates to the access rights for the individual element. Click to open the [Access rights dialog \[▶ 420\]](#). The setting is only available if a [user management \[▶ 393\]](#) was added to the PLC project. The following status messages are available:

Not set. All rights.	The default message is set, if the element is shown as available for all groups.
Rights are issued: Limited rights.	The message is set, if the element is shown with limited behavior for at least one group.

15.8.2.15 Radio button

A radio button enables configuration of any number of options via the  button, which can be arranged in one or several columns.




Properties editor

The properties of a visualization element - except [alignment and order \[▶ 377\]](#) - can all be configured in the [properties editor \[▶ 385\]](#). By default, this editor opens next to the visualization editor, or it can be opened explicitly via the "Properties" command (which can be found in the View menu as standard).

A property can be modified by editing the field "Value". To this end, an input field, a selection list, a dialog or checkbox that can be activated is provided in this field, depending on the element type. The value field opens

- after a double-click,
- after a single click in a selected field,
- via the space bar, if the field was already selected.

If a variable is assigned,

- simply enter its name.
- Use the  button to open the input assistant for selecting a variable. The Variables category lists all variables that have already been defined in the project.

Working in the list of properties can be made easier with the aid of default, sorting and filter functions.

Element properties

All element properties and their descriptions are listed below.

Element name	The element name can be changed. Standard name is "GenElemInst_x". "x" stands for a sequential number.
Element type	The element type is entered here. For three element groups it is possible to switch between the corresponding elements by changing the element type: <ul style="list-style-type: none"> • rectangle, rounded rectangle and ellipse [▶ 475] • Polygon, polyline or Bézier curve [▶ 490] • dip switch, power switch, push switch, push switch with LED, rocker switch [▶ 532]

Position

Here you can define the position (X/Y coordinates) and size (width and height) of the element in pixels. The origin is in the top left corner of the window. The positive x-axis is on the right, the positive y-axis runs downwards. If the values edited, the displayed element is simultaneously modified in the [visualization editor](#) [[▶ 376](#)].

X	Horizontal position in pixels – X=0 is the left edge of the window.
Y	Vertical position in pixels – Y=0 is the upper edge of the window.
Width	Width of the element in pixels
Height	Height of the element in pixels



Additional settings

Variable	Variable of type integer, which is used to store the index of the selected radio button.
Number of columns	Number of columns for arranging the radio buttons
Arrangement of the radio buttons	Select how the radio buttons are to be arranged: <ul style="list-style-type: none"> • From left to right: The radio buttons are arranged row by row, until all buttons have been positioned. • From top to bottom: The radio buttons are arranged column by column, until all buttons have been positioned.
Frame size	Size of the radio buttons Standard setting: From style [▶ 389]
Row height	Definition of the row height Standard setting: From style [▶ 389]

Text properties

Use of	The following two settings are available for selection: <ul style="list-style-type: none"> • Default style values • User-defined settings <ul style="list-style-type: none"> ◦ Horizontal alignment ◦ Vertical alignment ◦ Font ◦ Font color
--------	---

User-defined text properties


Horizontal alignment	<p>Defines the horizontal alignment of the text through selection of:</p> <ul style="list-style-type: none"> • Left • Centered • Right
Vertical alignment	<p>Defines the vertical alignment of the text through selection of:</p> <ul style="list-style-type: none"> • Top • Centered • Bottom
Font:	<p>Defines the font through selection from predefined fonts:</p> <ul style="list-style-type: none"> • Standard • Heading • Large heading • Title • Note <p>Press  to opens the dialog for user-defined font properties.</p>
Font color	<p>Defines the font color for the element. Either from the selection list or via the dialog that opens when  is clicked.</p>

State variables

These are dynamic definitions of the availability of the element in online mode.

Invisibility	<p>Specification of a boolean variable. If this returns TRUE, the element is invisible in online mode.</p>
Input disabled	<p>Specification of a boolean variable. If TRUE is returned, inputs for the element have no effect. Also, the element itself is greyed out in the visualization, to indicate that no user inputs are possible.</p> <p>If the visualization uses the user management [▶ 393], the elements for user groups with access right "only visible" are greyed out.</p>

Radio button settings

<p>Radio button</p> <ul style="list-style-type: none"> • Areas <ul style="list-style-type: none"> ◦ [n] 	<p>Click on the  button to create a new radio button in the editor. A further section is then listed in the properties editor. For each radio button an area is created that covers the corresponding settings.</p> <p>[n]: The number indexes the area. Clicking Delete deletes the corresponding radio button and its settings.</p>
---	---

Sections [n]

Text	<p>Button name</p> <p>Default: Radio button</p>
Tooltip	<p>Text to be shown as a tooltip</p>
Row spacing in pixels	<p>Distance to the top button in pixels</p>

Access rights

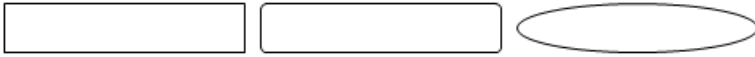
This setting relates to the access rights for the individual element. Click to open the [Access rights dialog \[▶ 420\]](#). The setting is only available if a [user management \[▶ 393\]](#) was added to the PLC project. The following status messages are available:

Not set. All rights.	The default message is set, if the element is shown as available for all groups.
Rights are issued: Limited rights.	The message is set, if the element is shown with limited behavior for at least one group.

15.8.3 Basic

15.8.3.1 Rectangle, rounded rectangle and ellipse

The visualization elements rectangle, rounded rectangle and ellipse are of the same element type. They can only be converted to each other only by changing the property Element type.




Properties editor

The properties of a visualization element - except alignment and order [[▶ 377](#)] - can all be configured in the properties editor [[▶ 385](#)]. By default, this editor opens next to the visualization editor, or it can be opened explicitly via the "Properties" command (which can be found in the View menu as standard).

A property can be modified by editing the field "Value". To this end, an input field, a selection list, a dialog or checkbox that can be activated is provided in this field, depending on the element type. The value field opens

- after a double-click,
- after a single click in a selected field,
- via the space bar, if the field was already selected.

If a variable is assigned,

- simply enter its name.
- Use the  button to open the input assistant for selecting a variable. The Variables category lists all variables that have already been defined in the project.

Working in the list of properties can be made easier with the aid of default, sorting and filter functions.

Element properties

All element properties and their descriptions are listed below.


Element name	The element name can be changed. Standard name is "GenElemInst_x". "x" stands for a sequential number.
Element type	The element type is entered here. For three element groups it is possible to switch between the corresponding elements by changing the element type: <ul style="list-style-type: none"> • <u>rectangle, rounded rectangle and ellipse</u> [▶ 475] • <u>Polygon, polyline or Bézier curve</u> [▶ 490] • <u>dip switch, power switch, push switch, push switch with LED, rocker switch</u> [▶ 532]

Position

Here you can define the position (X/Y coordinates) and size (width and height) of the element in pixels. The origin is in the top left corner of the window. The positive x-axis is on the right, the positive y-axis runs downwards. If the values edited, the displayed element is simultaneously modified in the visualization editor [[▶ 376](#)].

X	Horizontal position in pixels – X=0 is the left edge of the window.
Y	Vertical position in pixels – Y=0 is the upper edge of the window.
Width	Width of the element in pixels
Height	Height of the element in pixels

Center

When a value is edited, the corresponding element  is simultaneous moved in the [visualization editor](#) [[▶ 376](#)].

X	Horizontal position of the element pivot in pixels
Y	Vertical position of the element pivot in pixels

Colors

A color is defined based on a hexadecimal number consisting of red / green / blue (RGB) components. For each of these three colors, 256 (0-255) values are available, which can be entered here statically. The color can be selected from a selection list or via the color selection dialog, which can be opened via the button



. In addition, the level of transparency can be set for each color (0: fully transparent, 255: fully opaque).

Normal state • Frame color • Fill color	Select a frame and fill color for the default state. If the color switching variable is defined as FALSE, the element is in default state.
Alarm state • Frame color • Fill color	Select a frame and fill color for the element in alarm state. This is triggered, if the color switching variable is defined as TRUE.
Use color gradient	The checkbox is unticked by default. If it is ticked, the corresponding element is drawn with a color gradient.
Color gradient selection	The Gradient editor [▶ 405] opens.



Color gradient and transparency are not supported under Windows CE.

Appearance

The settings under Appearance are static definitions for the frame and the element filling.

Line width	Defines the frame line width in pixels. 0 codes the same as 1 and sets the line width to 1 pixel. If no frame is required, set the line type to invisible.
Fill type	Defines the fill type for the fill color: • Filled: The fill color is visible • Invisible: The fill color is invisible
Line type	Defines one of the following line types for the outline: • solid • dashed • dotted • dotdash • dash dot dot • Invisible: outline is invisible.



Texts

These properties are used for a static definition of the element labelling. Each can contain a formatting sequence [▶ 617], e.g. %s. In online mode the sequence is replaced by the contents of the variables defined in "Text variables".

Text	Enter a text. It is used to label the element. The text can contain a formatting sequence, e.g. %s. The corresponding variable is defined in "Text variables". Note: The text can also be entered directly. If the element is selected in the visualization editor, an input field can be opened by pressing the space bar.
Tooltip	Enter a text. It is used as a tooltip for the element and appears in the visualization only in online mode when the cursor is placed over an element. The text can contain a formatting sequence, e.g. %s. The corresponding variable is defined in "Text variables".




Text properties

These properties are used for a static definition of the font. A dynamic definition of the font is possible in the category "Font variables".

Horizontal alignment	Defines the horizontal alignment of the text through selection of: <ul style="list-style-type: none"> • Left • Centered • Right
Vertical alignment	Defines the vertical alignment of the text through selection of: <ul style="list-style-type: none"> • Top • Centered • Bottom
Text format	Defines the display of a text that is too long to be displayed completely in an element: <ul style="list-style-type: none"> • Default – the text extends beyond the element. • Line break – the text is automatically wrapped. • Ellipsis - the text is displayed as far as possible and then truncated with "...".
Font:	Defines the font through selection from predefined fonts: <ul style="list-style-type: none"> • Standard • Heading • Large heading • Title • Note <p>Press  to opens the dialog for user-defined font properties.</p>
Font color	Defines the font color for the element. Either from the selection list or via the dialog that opens when  is clicked.

Absolute movement

The element can be moved by changing the x- and y-position (pixels) of the top left corner of the element via an integer variable. Absolute coordinate values are used here.


<p>Motion</p> <ul style="list-style-type: none"> • X • Y 	<p>X: The integer variable entered here defines the current x-position of the top left corner of the element (in pixels). It can be used to move the element in x-direction. (A positive value moves the element from left to right).</p> <p>Y: The integer variable entered here defines the current y-position of the top left corner of the element (in pixels). It can be used to move the element in y-direction. (A positive value moves the element from top to bottom).</p>
<p>Rotation</p>	<p>The integer variable entered here defines the angle (angular degrees) for rotating the element around a rotation point.</p> <p>Positive values = clockwise</p> <p>Note: In contrast to the behavior with 'internal rotation' (see below), the element itself does not rotate. Click on the element to show the rotation point . It can be moved by pressing and holding the mouse button.</p>
<p>Scaling</p>	<p>The integer variable entered here defines the current scaling factor (percent). The element size is adjusted linearly according to this value. The value is implicitly divided by 1000, so that it is not necessary to use REAL variables in order to shrink the element. The scaling always refers to the rotation point (center). Click on the element to show the rotation point . It can be moved by pressing and holding the mouse button.</p>
<p>Internal rotation</p>	<p>The integer variable entered here defines the angle (angular degrees) by which the element is rotated around its rotation point; positive values=mathematically positive=clockwise. In contrast to "Rotation" (see above), the element itself rotates. Click on the element to show the rotation point (center) . It can be moved by pressing and holding the mouse button.</p>

Relative movement

The element can be moved relative to its fixed position. The top left and bottom right edges of the element are moved in x- or y-direction by a value (pixels) defined by an integer variable. In contrast to an absolute movement, a relative position is defined, i.e. the distance to the original position. This function can be used to change the shape of the element. Positive values move the horizontal edges downwards and/or the vertical edges to the right.

<p>Top left movement</p> <ul style="list-style-type: none"> • X • Y 	<ul style="list-style-type: none"> • X: Integer variable, the value of which indicates the number of pixels by which the top left corner is moved in x-direction. • Y: Integer variable, the value of which indicates the number of pixels by which the top left corner is moved in y-direction.
<p>Bottom right movement</p> <ul style="list-style-type: none"> • X • Y 	<ul style="list-style-type: none"> • X: Integer variable, the value of which indicates the number of pixels by which the bottom right corner is moved in x-direction. • Y: Integer variable, the value of which indicates the number of pixels by which the bottom right corner is moved in y-direction.

Text variables

You can display text that is stored in a variable. To do this, first add a formatting sequence in the text that is defined under the "Texts" property. Then assign a variable. In online mode the [formatting sequence](#)  6171 is replaced by the contents of the variables.

Example %f:

Enter "Result: %2.5f" in the "Texts" property. Enter "fValue" in the "Text variables" property. The variable must be a defined IEC variable. The element will then be labeled in online mode with "Result: 12.12345" if fValue = 12.1234567.

<p>Text variable</p>	<p>Variable (of default data type) containing the information to be displayed. The type must match the formatting sequence in the "Texts" setting.</p>
<p>Tooltipvariable</p>	<p>Variable of type String containing the tooltip text to be displayed. The entry in the property "Texts" must contain a formatting sequence.</p>

Dynamic texts

These parameters are used to define dynamic texts originating from [text lists \[► 138\]](#). This enables [language change \[► 616\]](#), for example.

A further possibility to define a text dynamically is to supply the text via a string variable. (see the category "Text variables")

Text list	Name of the text list, as used in the project tree, as a string Example: 'TL_ErrorList'
Textindex	Index (ID) of the text, as defined in the text list, as a string. It can be specified directly in a static manner or as a string variable.
Tooltipindex	Index (ID) of the tooltip text, as defined in the text list, as a string. It can be specified directly in a static manner or as a string variable.

Font variables

These variables are used for dynamic font definitions for element texts via project variables. Static definitions are configured under "Text properties".

Font name	Specification of a variable of the type String that contains the font name that is to be used to label the element. (name specified as in the standard font dialog) Example: MAIN.sFont (sFont := 'Arial');
Size	Variable of type INT containing the size of the element text in pixels, as in the default dialog 'Font'. Example: MAIN.nHeight (nHeight := 16;)
Flags	Variable of type DWORD for defining the font display via one of the flag values listed below. A combined definition can be achieved by adding the respective flag values and specifying the sum. <ul style="list-style-type: none"> • 1: italics • 2: bold • 4: underlined • 8: deleted Example: MAIN.nFlag (nFlag := 6;) The text is shown bold and underlined.
Character set	The character set to be used for the font can be defined via the default font number. This number can be specified via a DWORD variable (see also the definition in the default font dialog)
Color	Variable of type DWORD for defining the color of the element text.
Flags for the text alignment	Variable of type DWORD for defining the text alignment via one of the flag values listed below. A combined definition can be achieved by adding the respective flag values and specifying the sum. <ul style="list-style-type: none"> • 0: top left • 1: centered horizontally • 2: right • 4: centered vertically • 8: below Example MAIN.nFlag (nFlag := 5;) The text is displayed centered horizontally and vertically.

Color variables

The color variables used for dynamic definition of the element colors via project variables of type DWORD. A color is defined based on a hexadecimal number consisting of red, green and blue (RGB) components. In addition, the variables are used to specify the transparency of the color (FF: fully opaque - 00: fully transparent). The DWORD has the following structure: 16#TTRRGGBB

i The structure of the hexadecimal number differs, compared to TwinCAT 2. In TwinCAT 3, in addition to the RGB components, the color transparency can also be defined with the hexadecimal number. The transparency is indicated by the first two digits after "16#". Colors with definitions starting with "16#00" are invisible, since they are fully transparent.

Example:

nFillColor := 16#FF8FE03F;

- FF: transparency (fully opaque)
- 8F: red
- E0: green
- 3F: blue

Color change	Boolean variable, which controls the switching of the element color between "normal state" (variable = FALSE) and "alarm state" (variable = TRUE).
Normal state • Frame color • Fill color	Variables of type DWORD for defining the frame and fill color for the element. They overwrite the values that are currently defined in "Colors". The values in the project variable are used, if the variable defined in "Color change" is FALSE.
Alarm state • Frame color • Fill color	Variables of type DWORD for defining the frame and fill color for the element in alarm state. They overwrite the values that are currently defined in "Colors". The values in the project variable are used, if the variable defined in "Color change" is TRUE.

i Transparency is not supported under Windows CE.

Appearance variables

To be used for a dynamic definition of the contour appearance and the element filling. Static definitions are specified under "Appearance".

Line width	Variable of type integer for defining the line width of the element in pixels. It overwrites the fixed value that was specified under "Appearance".
Fill type	Variable of type DWORD for defining the element filling. The color specified under color variables can be displayed or ignored: <ul style="list-style-type: none"> • Variable value = 0: filled • Variable value > 0: invisible, i.e. no filling is visible
Line type	Variable of type DWORD for defining the contour. The following values correspond to the following line type: <ul style="list-style-type: none"> • 0: solid • 1: dashed • 2: dotted • 3: dotdash • 4: dash dot dot • 8: invisible. That is, the element is shown without a contour.

State variables

These are dynamic definitions of the availability of the element in online mode.

Invisibility	Specification of a boolean variable. If this returns TRUE, the element is invisible in online mode.
Input disabled	Specification of a boolean variable. If TRUE is returned, inputs for the element have no effect. Also, the element itself is greyed out in the visualization, to indicate that no user inputs are possible. If the visualization uses the user management [▶ 393] , the elements for user groups with access right "only visible" are grayed out.

Input configuration

Here you can define the consequential action that should be performed when the user makes an input in the element in online mode. As long as no follow-on actions are defined, "Configure..." appears in the Properties field. Click on "Configure..." to open the [Input Configuration \[▶ 406\]](#), where you can assign follow-on actions. Each input action can be assigned any number of follow-on actions.

The following input events are available for an element:

- OnDialogClosed
- OnMouseClicked
- OnMouseDown
- OnMouseEnter
- OnMouseLeave
- OnMouseMove
- OnMouseUp

OnDialogClosed	This event is triggered if one of the dialog boxes that were opened for user inputs is closed within a visualization. Note: This property is not limited to the element, for which it is configured, but applies within the entire visualization. It therefore does not respond to each dialog closing action. Currently there is no way to define such a property for the entire visualization. It therefore has to be assigned to one of its elements.
OnMouseClicked	This mouse event is triggered when the cursor points to an element and a full mouse click (pressing and releasing the mouse button) is executed on this item.
OnMouseDown	This mouse event is triggered when the mouse button is pressed while the cursor points to an element. It is irrelevant where on the visualization the mouse button is released again.
OnMouseEnter	This mouse event is triggered when the cursor is dragged to the element. It is irrelevant whether the mouse button is pressed or released.
OnMouseLeave	This mouse event is triggered when the cursor leaves the element. It is irrelevant whether the mouse button is pressed or released.
OnMouseMove	This mouse event is triggered when the cursor is moved within the element. It is irrelevant whether the mouse button is pressed or released.
OnMouseUp	This mouse event occurs when the mouse button is released on the element. The mouse button was pressed prior to that outside of the element.

Input configuration - hotkeys

A hotkey can be used to define a key or key combination and link it with a follow-on action (e.g. MouseDown, MouseUp), which is to be executed when a key event occurs (KeyDown, KeyUp). By default the MouseDown action is executed on KeyDown (press key) and the MouseUp action on KeyUp (release key). This can be useful if a visualization is to be operated both via mouse actions and keyboard entries, since the input actions then only have to be configured once. The key configuration for an element is also managed in the [hotkeys configuration \[▶ 392\]](#) for the visualization. Any changes are always synchronized between this and the element properties editor.

Key	Assigning a key. A selection list contains all currently supported keys, e.g. M.
Event(s)	Definition of the event to be executed, if the key or key and modifier are used. Possible values that are available in a selection list: <ul style="list-style-type: none"> • No action • MouseDown action when the key is pressed • MouseUp action when the key is released • MouseDown/MouseUp action when the key is pressed/released
Shift	If this option is enabled, the key has to be used in combination with the Shift key.
Control	If this option is enabled, the key has to be used in combination with the Ctrl key.
Alt	If this option is enabled, the key has to be used in combination with the Alt key.

Input configuration – Keys

"Keys" can be used to specify that the value of a boolean project variable is set depending on the mouse behavior for the event "Keys".

Variable	Boolean variable whose value is TRUE if the mouse button is pressed while the cursor points to the element. The value becomes FALSE again when the mouse button is released or the cursor leaves the element.
FALSE keys	If this option is enabled, the key behavior described above is reversed for the corresponding variable. That is, when the mouse button pressed, the variable value is set to FALSE. When the button is, the variable value is set to TRUE.
Switch on enter if mouse trapped	If this option is enabled, the variable value behaves as described under 'Variables', as long as the cursor is within the element area. If the cursor leaves the element area while the mouse button is pressed, the variable is set to FALSE. However, it is automatically set to TRUE again, if the cursor returns to the element area without the mouse button being released. In other words, the system takes into account that the mouse is "trapped" as long as the mouse button is pressed, even if the cursor leaves the element area.

Input configuration - Switching

"Switching" can be used to specify that for this event the value of a boolean project variable is set depending on the mouse behavior.

Variable	Boolean variable whose value switches between TRUE and FALSE with each mouse click on the element. No switching occurs if the cursor is moved away from the element while the mouse button is pressed. Can be used to cancel a switching action.
Switch on release, if mouse trapped	If this option is enabled, the variable value behaves as described under 'Variables', as long as the cursor is within the element area. If the cursor leaves the element area while the mouse button is pressed and the mouse button is then released, the variable is switched anyway. In other words, the system takes into account that the mouse is "trapped" as long as the mouse button is pressed, even if the cursor leaves the element area.

Access rights

This setting relates to the access rights for the individual element. Click to open the [Access rights dialog \[► 420\]](#). The setting is only available if a [user management \[► 393\]](#) was added to the PLC project. The following status messages are available:

Not set. All rights.	The default message is set, if the element is shown as available for all groups.
Rights are issued: Limited rights.	The message is set, if the element is shown with limited behavior for at least one group.

15.8.3.2 Line

The element 'Line' is a simple line that is characteristically described by two defined points.



Properties editor

The properties of a visualization element - except alignment and order [▶ 377] - can all be configured in the properties editor [▶ 385]. By default, this editor opens next to the visualization editor, or it can be opened explicitly via the "Properties" command (which can be found in the View menu as standard).

A property can be modified by editing the field "Value". To this end, an input field, a selection list, a dialog or checkbox that can be activated is provided in this field, depending on the element type. The value field opens

- after a double-click,
- after a single click in a selected field,
- via the space bar, if the field was already selected.

If a variable is assigned,

- simply enter its name.
- Use the button to open the input assistant for selecting a variable. The Variables category lists all variables that have already been defined in the project.

Working in the list of properties can be made easier with the aid of default, sorting and filter functions.

Element properties

All element properties and their descriptions are listed below.

Element name	The element name can be changed. Standard name is "GenElemInst_x". "x" stands for a sequential number.
Element type	The element type is entered here. For three element groups it is possible to switch between the corresponding elements by changing the element type: <ul style="list-style-type: none"> • <u>rectangle, rounded rectangle and ellipse</u> [▶ 475] • <u>Polygon, polyline or Bézier curve</u> [▶ 490] • <u>dip switch, power switch, push switch, push switch with LED, rocker switch</u> [▶ 532]


Position

The positions that are characteristic for the visualization element (X/Y coordinates) have to be defined in pixels. The origin is in the top left corner of the window. The positive x-axis is on the right, the positive y-axis runs downwards. If the values are edited, the corresponding element is simultaneously modified in the visualization editor [▶ 376].

The number of points through which the element is defined can be increased or reduced. Selecting one of the points in the visualization editor while pressing the Ctrl key results in this point being duplicated. This automatically adds an entry for a further point in the properties. To delete a point, select it while pressing the Ctrl and Shift keys at the same time. The corresponding entry is then deleted from the properties.

[0] X / Y [n] X / Y	X/Y positions (in pixels) of the individual element points [0] is the position of the start point. The following points are numbered consecutively.
X	Horizontal position in pixels. X=0 is the left-hand edge of the window.
Y	Vertical position in pixels. Y=0 is the top edge of the window.

Center

When a value is edited, the corresponding element  is simultaneous moved in the [visualization editor](#) [[▶ 376](#)].

X	Horizontal position of the element pivot in pixels
Y	Vertical position of the element pivot in pixels

Colors

A color is defined based on a hexadecimal number consisting of red / green / blue (RGB) components. For each of these three colors, 256 (0-255) values are available, which can be entered here statically. The color can be selected from a selection list or via the color selection dialog, which can be opened via the button



. In addition, the level of transparency can be set for each color (0: fully transparent, 255: fully opaque).

Color	Select a frame and fill color for the default state. If the color switching variable is defined as FALSE, the element is in default state.
Alarm color	Select a frame and fill color for the element in alarm state. This is triggered, if the color switching variable is defined as TRUE.



Transparency is not supported under Windows CE.

Appearance

Line width	Defines the frame line width in pixels. 0 codes the same as 1 and sets the line width to 1 pixel. If no frame is required, set the line type to invisible.
Line type	Defines one of the following line types for the outline: <ul style="list-style-type: none"> • solid • dashed • dotted • dotdash • dash dot dot • Invisible: outline is invisible.



Texts

These properties are used for a static definition of the element labelling. Each can contain a [formatting sequence](#) [[▶ 617](#)], e.g. %s. In online mode the sequence is replaced by the contents of the variables defined in "Text variables".

Text	Enter a text. It is used to label the element. The text can contain a formatting sequence, e.g. %s. The corresponding variable is defined in "Text variables". Note: The text can also be entered directly. If the element is selected in the visualization editor, an input field can be opened by pressing the space bar.
Tooltip	Enter a text. It is used as a tooltip for the element and appears in the visualization only in online mode when the cursor is placed over an element. The text can contain a formatting sequence, e.g. %s. The corresponding variable is defined in "Text variables".




Text properties

These properties are used for a static definition of the font. A dynamic definition of the font is possible in the category "Font variables".

Horizontal alignment	<p>Defines the horizontal alignment of the text through selection of:</p> <ul style="list-style-type: none"> • Left • Centered • Right
Vertical alignment	<p>Defines the vertical alignment of the text through selection of:</p> <ul style="list-style-type: none"> • Top • Centered • Bottom
Text format	<p>Defines the display of a text that is too long to be displayed completely in an element:</p> <ul style="list-style-type: none"> • Default – the text extends beyond the element. • Line break – the text is automatically wrapped. • Ellipsis - the text is displayed as far as possible and then truncated with "...".
Font:	<p>Defines the font through selection from predefined fonts:</p> <ul style="list-style-type: none"> • Standard • Heading • Large heading • Title • Note <p>Press  to opens the dialog for user-defined font properties.</p>
Font color	<p>Defines the font color for the element. Either from the selection list or via the dialog that opens when  is clicked.</p>

Absolute movement

The element can be moved by changing the x- and y-position (pixels) of the top left corner of the element via an integer variable. Absolute coordinate values are used here.

<p>Motion</p> <ul style="list-style-type: none"> • X • Y 	<p>X: The integer variable entered here defines the current x-position of the top left corner of the element (in pixels). It can be used to move the element in x-direction. (A positive value moves the element from left to right).</p> <p>Y: The integer variable entered here defines the current y-position of the top left corner of the element (in pixels). It can be used to move the element in y-direction. (A positive value moves the element from top to bottom).</p>
Rotation	<p>The integer variable entered here defines the angle (angular degrees) for rotating the element around a rotation point.</p> <p>Positive values = clockwise</p> <p>Note: In contrast to the behavior with 'internal rotation' (see below), the element itself does not rotate. Click on the element to show the rotation point . It can be moved by pressing and holding the mouse button.</p>
Scaling	<p>The integer variable entered here defines the current scaling factor (percent). The element size is adjusted linearly according to this value. The value is implicitly divides by 1000, so that it is not necessary to use REAL variables in order to shrink the element. The scaling always refers to the rotation point (center). Click on the element to show the rotation point . It can be moved by pressing and holding the mouse button.</p>
Internal rotation	<p>The integer variable entered here defines the angle (angular degrees) by which the element is rotated around its rotation point; positive values=mathematically positive=clockwise. In contrast to "Rotation" (see above), the element itself rotates. Click on the element to show the rotation point (center) . It can be moved by pressing and holding the mouse button.</p>

Relative movement

The element can be moved relative to its fixed position. The top left and bottom right edges of the element are moved in x- or y-direction by a value (pixels) defined by an integer variable. In contrast to an absolute movement, a relative position is defined, i.e. the distance to the original position. This function can be used to change the shape of the element. Positive values move the horizontal edges downwards and/or the vertical edges to the right.

Movement point [n] <ul style="list-style-type: none"> • X • Y 	<ul style="list-style-type: none"> • X: Integer variable, the value of which indicates the number of pixels by which the point[0] / point[1] is moved in x-direction. • Y: Integer variable, the value of which indicates the number of pixels by which the point[0] / point[1] is moved in y-direction.
---	--

Text variables

You can display text that is stored in a variable. To do this, first add a formatting sequence in the text that is defined under the "Texts" property. Then assign a variable. In online mode the [formatting sequence \[▶ 617\]](#) is replaced by the contents of the variables.

Example %f:

Enter "Result: %2.5f" in the "Texts" property. Enter "fValue" in the "Text variables" property. The variable must be a defined IEC variable. The element will then be labeled in online mode with "Result: 12.12345" if fValue = 12.1234567.

Text variable	Variable (of default data type) containing the information to be displayed. The type must match the formatting sequence in the "Texts" setting.
Tooltipvariable	Variable of type String containing the tooltip text to be displayed. The entry in the property "Texts" must contain a formatting sequence.

Dynamic texts

These parameters are used to define dynamic texts originating from [text lists \[▶ 138\]](#). This enables [language change \[▶ 616\]](#), for example.

A further possibility to define a text dynamically is to supply the text via a string variable. (see the category "Text variables")

Text list	Name of the text list, as used in the project tree, as a string Example: 'TL_ErrorList'
Textindex	Index (ID) of the text, as defined in the text list, as a string. It can be specified directly in a static manner or as a string variable.
Tooltipindex	Index (ID) of the tooltip text, as defined in the text list, as a string. It can be specified directly in a static manner or as a string variable.

Font variables

These variables are used for dynamic font definitions for element texts via project variables. Static definitions are configured under "Text properties".

Font name	Specification of a variable of the type String that contains the font name that is to be used to label the element. (name specified as in the standard font dialog) Example: MAIN.sFont (sFont := 'Arial');
Size	Variable of type INT containing the size of the element text in pixels, as in the default dialog 'Font'. Example: MAIN.nHeight (nHeight := 16;)
Flags	Variable of type DWORD for defining the font display via one of the flag values listed below. A combined definition can be achieved by adding the respective flag values and specifying the sum. <ul style="list-style-type: none"> • 1: italics • 2: bold • 4: underlined • 8: deleted Example: MAIN.nFlag (nFlag := 6;) The text is shown bold and underlined.
Character set	The character set to be used for the font can be defined via the default font number. This number can be specified via a DWORD variable (see also the definition in the default font dialog)
Color	Variable of type DWORD for defining the color of the element text.
Flags for the text alignment	Variable of type DWORD for defining the text alignment via one of the flag values listed below. A combined definition can be achieved by adding the respective flag values and specifying the sum. <ul style="list-style-type: none"> • 0: top left • 1: centered horizontally • 2: right • 4: centered vertically • 8: below Example MAIN.nFlag (nFlag := 5;) The text is displayed centered horizontally and vertically.

Color variables

The color variables used for dynamic definition of the element colors via project variables of type DWORD. A color is defined based on a hexadecimal number consisting of red, green and blue (RGB) components. In addition, the variables are used to specify the transparency of the color (FF: fully opaque - 00: fully transparent). The DWORD has the following structure: 16#TTRRGGBB



The structure of the hexadecimal number differs from TwinCAT 2. In TwinCAT 3 it is possible to define the transparency of the color with the hexadecimal number in addition to the RGB proportions. The transparency is indicated by the first two digits after "16#". Colors with definitions starting with "16#00" are invisible, since they are fully transparent.

Example:

nFillColor := 16#FF8FE03F;

- FF: transparency (fully opaque)
- 8F: red
- E0: green
- 3F: blue

Color change	Boolean variable, which controls the switching of the element color between "normal state" (variable = FALSE) and "alarm state" (variable = TRUE).
Normal state	Variable of type DWORD for defining the element color. It overwrites the value that is currently defined in "Colors". The value in the project variable is used, if the variable defined in "Color change" is FALSE.
Alarm state	Variable of type DWORD for defining the element color in alarm state. It overwrites the value that is currently defined in "Colors". The value in the project variable is used, if the variable defined in "Color change" is TRUE.



Transparency is not supported under Windows CE.

State variables

These are dynamic definitions of the availability of the element in online mode.

Invisibility	Specification of a boolean variable. If this returns TRUE, the element is invisible in online mode.
Input disabled	Specification of a boolean variable. If TRUE is returned, inputs for the element have no effect. Also, the element itself is greyed out in the visualization, to indicate that no user inputs are possible. If the visualization uses the user management [▶ 393] , the elements for user groups with access right "only visible" are grayed out.

Line type variable

Dynamic definition the line type via a variable.

Integer value	Integer variable for specifying data types. The value defines the line type. The following variable values are supported: <ul style="list-style-type: none"> • 0 = solid • 1 = dashed • 2 = dotted • 3 = dotdash • 4 = dash dot dot • 8 = invisible: The line is invisible. This overwrites the fixed value specified in the category "Appearance".
---------------	---

Line width variable

Dynamic definition of the width of a line element via a variable.

Integer value	Variable of type integer for defining the line width of the element in pixels. This overwrites the fixed value specified in the category "Appearance". Not that a code of 0 has the same effect as 1, and the line width is set to 1 pixel.
---------------	---

Input configuration

Here you can define the consequential action that should be performed when the user makes an input in the element in online mode. As long as no follow-on actions are defined, "Configure..." appears in the Properties field. Click on "Configure..." to open the [Input Configuration \[▶ 406\]](#), where you can assign follow-on actions. Each input action can be assigned any number of follow-on actions.

The following input events are available for an element:

- OnDialogClosed
- OnMouseClicked

- OnMouseDown
- OnMouseEnter
- OnMouseLeave
- OnMouseMove
- OnMouseUp

OnDialogClosed	This event is triggered if one of the dialog boxes that were opened for user inputs is closed within a visualization. Note: This property is not limited to the element, for which it is configured, but applies within the entire visualization. It therefore does not respond to each dialog closing action. Currently there is no way to define such a property for the entire visualization. It therefore has to be assigned to one of its elements.
OnMouseClicked	This mouse event is triggered when the cursor points to an element and a full mouse click (pressing and releasing the mouse button) is executed on this item.
OnMouseDown	This mouse event is triggered when the mouse button is pressed while the cursor points to an element. It is irrelevant where on the visualization the mouse button is released again.
OnMouseEnter	This mouse event is triggered when the cursor is dragged to the element. It is irrelevant whether the mouse button is pressed or released.
OnMouseLeave	This mouse event is triggered when the cursor leaves the element. It is irrelevant whether the mouse button is pressed or released.
OnMouseMove	This mouse event is triggered when the cursor is moved within the element. It is irrelevant whether the mouse button is pressed or released.
OnMouseUp	This mouse event occurs when the mouse button is released on the element. The mouse button was pressed prior to that outside of the element.

Input configuration - hotkeys

A hotkey can be used to define a key or key combination and link it with a follow-on action (e.g. MouseDown, MouseUp), which is to be executed when a key event occurs (KeyDown, KeyUp). By default the MouseDown action is executed on KeyDown (press key) and the MouseUp action on KeyUp (release key). This can be useful if a visualization is to be operated both via mouse actions and keyboard entries, since the input actions then only have to be configured once. The key configuration for an element is also managed in the [hotkeys configuration \[► 392\]](#) for the visualization. Any changes are always synchronized between this and the element properties editor.

Key	Assigning a key. A selection list contains all currently supported keys, e.g. M.
Event(s)	Definition of the event to be executed, if the key or key and modifier are used. Possible values that are available in a selection list: <ul style="list-style-type: none"> • No action • MouseDown action when the key is pressed • MouseUp action when the key is released • MouseDown/MouseUp action when the key is pressed/released
Shift	If this option is enabled, the key has to be used in combination with the Shift key.
Control	If this option is enabled, the key has to be used in combination with the Ctrl key.
Alt	If this option is enabled, the key has to be used in combination with the Alt key.

Input configuration – Keys

"Keys" can be used to specify that the value of a boolean project variable is set depending on the mouse behavior for the event "Keys".

Variable	Boolean variable whose value is TRUE if the mouse button is pressed while the cursor points to the element. The value becomes FALSE again when the mouse button is released or the cursor leaves the element.
FALSE keys	If this option is enabled, the key behavior described above is reversed for the corresponding variable. That is, when the mouse button pressed, the variable value is set to FALSE. When the button is, the variable value is set to TRUE.
Switch on enter if mouse trapped	If this option is enabled, the variable value behaves as described under 'Variables', as long as the cursor is within the element area. If the cursor leaves the element area while the mouse button is pressed, the variable is set to FALSE. However, it is automatically set to TRUE again, if the cursor returns to the element area without the mouse button being released. In other words, the system takes into account that the mouse is "trapped" as long as the mouse button is pressed, even if the cursor leaves the element area.

Input configuration - Switching

"Switching" can be used to specify that for this event the value of a boolean project variable is set depending on the mouse behavior.

Variable	Boolean variable whose value switches between TRUE and FALSE with each mouse click on the element. No switching occurs if the cursor is moved away from the element while the mouse button is pressed. Can be used to cancel a switching action.
Switch on release, if mouse trapped	If this option is enabled, the variable value behaves as described under 'Variables', as long as the cursor is within the element area. If the cursor leaves the element area while the mouse button is pressed and the mouse button is then released, the variable is switched anyway. In other words, the system takes into account that the mouse is "trapped" as long as the mouse button is pressed, even if the cursor leaves the element area.

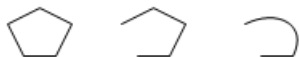
Access rights

This setting relates to the access rights for the individual element. Click to open the [Access rights dialog \[► 420\]](#). The setting is only available if a [user management \[► 393\]](#) was added to the PLC project. The following status messages are available:

Not set. All rights.	The default message is set, if the element is shown as available for all groups.
Rights are issued: Limited rights.	The message is set, if the element is shown with limited behavior for at least one group.

15.8.3.3 Polygon, polyline or Bézier curve

Polygon, polyline and Bézier curve are of the same element type. They can only be converted to each other only by changing the property Element type.




Properties editor

The properties of a visualization element - except [alignment and order \[► 377\]](#) - can all be configured in the [properties editor \[► 385\]](#). By default, this editor opens next to the visualization editor, or it can be opened explicitly via the "Properties" command (which can be found in the View menu as standard).

A property can be modified by editing the field "Value". To this end, an input field, a selection list, a dialog or checkbox that can be activated is provided in this field, depending on the element type. The value field opens

- after a double-click,
- after a single click in a selected field,
- via the space bar, if the field was already selected.

If a variable is assigned,

- simply enter its name.
- Use the  button to open the input assistant for selecting a variable. The Variables category lists all variables that have already been defined in the project.

Working in the list of properties can be made easier with the aid of default, sorting and filter functions.

Element properties

All element properties and their descriptions are listed below.

Element name	The element name can be changed. Standard name is "GenElemInst_x". "x" stands for a sequential number.
Element type	The element type is entered here. For three element groups it is possible to switch between the corresponding elements by changing the element type: <ul style="list-style-type: none"> • rectangle, rounded rectangle and ellipse [▶ 475] • Polygon, polyline or Bézier curve [▶ 490] • dip switch, power switch, push switch, push switch with LED, rocker switch [▶ 532]


Position

The positions that are characteristic for the visualization element (X/Y coordinates) have to be defined in pixels. The origin is in the top left corner of the window. The positive x-axis is on the right, the positive y-axis runs downwards. If the values are edited, the corresponding element is simultaneously modified in the [visualization editor \[▶ 376\]](#).

The number of points through which the element is defined can be increased or reduced. Selecting one of the points in the visualization editor while pressing the Ctrl key results in this point being duplicated. This automatically adds an entry for a further point in the properties. To delete a point, select it while pressing the Ctrl and Shift keys at the same time. The corresponding entry is then deleted from the properties.

[0] X / Y [n] X / Y	X/Y positions (in pixels) of the individual element points [0] is the position of the start point. The following points are numbered consecutively.
X	Horizontal position in pixels. X=0 is the left-hand edge of the window.
Y	Vertical position in pixels. Y=0 is the top edge of the window.

Center

When a value is edited, the corresponding element  is simultaneous moved in the [visualization editor \[▶ 376\]](#).

X	Horizontal position of the element pivot in pixels
Y	Vertical position of the element pivot in pixels

Colors

A color is defined based on a hexadecimal number consisting of red / green / blue (RGB) components. For each of these three colors, 256 (0-255) values are available, which can be entered here statically. The color can be selected from a selection list or via the color selection dialog, which can be opened via the button



. In addition, the level of transparency can be set for each color (0: fully transparent, 255: fully opaque).

Normal state • Frame color • Fill color	Select a frame and fill color for the default state. If the color switching variable is defined as FALSE, the element is in default state.
Alarm state • Frame color • Fill color	Select a frame and fill color for the element in alarm state. This is triggered, if the color switching variable is defined as TRUE.
Use color gradient	The checkbox is unticked by default. If it is ticked, the corresponding element is drawn with a color gradient.
Color gradient selection	The Gradient editor [▶ 405] opens.



Color gradient and transparency are not supported under Windows CE.

Appearance

The settings under Appearance are static definitions for the frame and the element filling.

Line width	Defines the frame line width in pixels. 0 codes the same as 1 and sets the line width to 1 pixel. If no frame is required, set the line type to invisible.
Fill type	Defines the fill type for the fill color: • Filled: The fill color is visible • Invisible: The fill color is invisible
Line type	Defines one of the following line types for the outline: • solid • dashed • dotted • dotdash • dash dot dot • Invisible: outline is invisible.



Texts

These properties are used for a static definition of the element labelling. Each can contain a [formatting sequence](#) [▶ 617], e.g. %s. In online mode the sequence is replaced by the contents of the variables defined in "Text variables".

Text	Enter a text. It is used to label the element. The text can contain a formatting sequence, e.g. %s. The corresponding variable is defined in "Text variables". Note: The text can also be entered directly. If the element is selected in the visualization editor, an input field can be opened by pressing the space bar.
Tooltip	Enter a text. It is used as a tooltip for the element and appears in the visualization only in online mode when the cursor is placed over an element. The text can contain a formatting sequence, e.g. %s. The corresponding variable is defined in "Text variables".




Text properties

These properties are used for a static definition of the font. A dynamic definition of the font is possible in the category "Font variables".

Horizontal alignment	<p>Defines the horizontal alignment of the text through selection of:</p> <ul style="list-style-type: none"> • Left • Centered • Right
Vertical alignment	<p>Defines the vertical alignment of the text through selection of:</p> <ul style="list-style-type: none"> • Top • Centered • Bottom
Text format	<p>Defines the display of a text that is too long to be displayed completely in an element:</p> <ul style="list-style-type: none"> • Default – the text extends beyond the element. • Line break – the text is automatically wrapped. • Ellipsis - the text is displayed as far as possible and then truncated with "...".
Font:	<p>Defines the font through selection from predefined fonts:</p> <ul style="list-style-type: none"> • Standard • Heading • Large heading • Title • Note <p>Press  to opens the dialog for user-defined font properties.</p>
Font color	<p>Defines the font color for the element. Either from the selection list or via the dialog that opens when  is clicked.</p>

Absolute movement

The element can be moved by changing the x- and y-position (pixels) of the top left corner of the element via an integer variable. Absolute coordinate values are used here.

<p>Motion</p> <ul style="list-style-type: none"> • X • Y 	<p>X: The integer variable entered here defines the current x-position of the top left corner of the element (in pixels). It can be used to move the element in x-direction. (A positive value moves the element from left to right).</p> <p>Y: The integer variable entered here defines the current y-position of the top left corner of the element (in pixels). It can be used to move the element in y-direction. (A positive value moves the element from top to bottom).</p>
Rotation	<p>The integer variable entered here defines the angle (angular degrees) for rotating the element around a rotation point.</p> <p>Positive values = clockwise</p> <p>Note: In contrast to the behavior with 'internal rotation' (see below), the element itself does not rotate. Click on the element to show the rotation point . It can be moved by pressing and holding the mouse button.</p>
Scaling	<p>The integer variable entered here defines the current scaling factor (percent). The element size is adjusted linearly according to this value. The value is implicitly divides by 1000, so that it is not necessary to use REAL variables in order to shrink the element. The scaling always refers to the rotation point (center). Click on the element to show the rotation point . It can be moved by pressing and holding the mouse button.</p>
Internal rotation	<p>The integer variable entered here defines the angle (angular degrees) by which the element is rotated around its rotation point; positive values=mathematically positive=clockwise. In contrast to "Rotation" (see above), the element itself rotates. Click on the element to show the rotation point (center) . It can be moved by pressing and holding the mouse button.</p>

Dynamic points

Dynamic definition of the points that define the element

Array of points	Variable pointing to an array with the structure VisuElems.VisuStructPoint. The VisuElems.VisuStructPoint components iX and iY contain the X/Y coordinates of a point. The VisuElems.VisuStructPoint components iX and iY contain the X/Y coordinates of a point. The number of element points must be declared explicitly in the following setting 'Number of points'. Example: pPoints : POINTER TO ARRAY[1..100] OF VisuElems.VisuStructPoint;
Number of points	Variable of type integer for defining the number of points of the element. Example: nCount : INT := 24; The element is defined through 24 individual points. This specification is necessary, since the individual points are defined via a pointer, and it is not possible to control the number.



In order to enable dynamic definition of points, either the [PLC HMI](#) [▶ 603], the [PLC HMI Web](#) [▶ 608] or both must be enabled.

Text variables

You can display text that is stored in a variable. To do this, first add a formatting sequence in the text that is defined under the "Texts" property. Then assign a variable. In online mode the [formatting sequence](#) [▶ 617] is replaced by the contents of the variables.

Example %f:

Enter "Result: %2.5f" in the "Texts" property. Enter "fValue" in the "Text variables" property. The variable must be a defined IEC variable. The element will then be labeled in online mode with "Result: 12.12345" if fValue = 12.1234567.

Text variable	Variable (of default data type) containing the information to be displayed. The type must match the formatting sequence in the "Texts" setting.
Tooltipvariable	Variable of type String containing the tooltip text to be displayed. The entry in the property "Texts" must contain a formatting sequence.

Dynamic texts

These parameters are used to define dynamic texts originating from [text lists](#) [▶ 138]. This enables [language change](#) [▶ 616], for example.

A further possibility to define a text dynamically is to supply the text via a string variable. (see the category "Text variables")

Text list	Name of the text list, as used in the project tree, as a string Example: 'TL_ErrorList'
Textindex	Index (ID) of the text, as defined in the text list, as a string. It can be specified directly in a static manner or as a string variable.
Tooltipindex	Index (ID) of the tooltip text, as defined in the text list, as a string. It can be specified directly in a static manner or as a string variable.

Font variables

These variables are used for dynamic font definitions for element texts via project variables. Static definitions are configured under "Text properties".

Font name	Specification of a variable of the type String that contains the font name that is to be used to label the element. (name specified as in the standard font dialog) Example: MAIN.sFont (sFont := 'Arial');
Size	Variable of type INT containing the size of the element text in pixels, as in the default dialog 'Font'. Example: MAIN.nHeight (nHeight := 16;)
Flags	Variable of type DWORD for defining the font display via one of the flag values listed below. A combined definition can be achieved by adding the respective flag values and specifying the sum. <ul style="list-style-type: none"> • 1: italics • 2: bold • 4: underlined • 8: deleted Example: MAIN.nFlag (nFlag := 6;) The text is shown bold and underlined.
Character set	The character set to be used for the font can be defined via the default font number. This number can be specified via a DWORD variable (see also the definition in the default font dialog)
Color	Variable of type DWORD for defining the color of the element text.
Flags for the text alignment	Variable of type DWORD for defining the text alignment via one of the flag values listed below. A combined definition can be achieved by adding the respective flag values and specifying the sum. <ul style="list-style-type: none"> • 0: top left • 1: centered horizontally • 2: right • 4: centered vertically • 8: below Example MAIN.nFlag (nFlag := 5;) The text is displayed centered horizontally and vertically.

Appearance variables

To be used for a dynamic definition of the contour appearance and the element filling. Static definitions are specified under "Appearance".

Line width	Variable of type integer for defining the line width of the element in pixels. It overwrites the fixed value that was specified under "Appearance".
Fill type	Variable of type DWORD for defining the element filling. The color specified under color variables can be displayed or ignored: <ul style="list-style-type: none"> • Variable value = 0: filled • Variable value > 0: invisible, i.e. no filling is visible
Line type	Variable of type DWORD for defining the contour. The following values correspond to the following line type: <ul style="list-style-type: none"> • 0: solid • 1: dashed • 2: dotted • 3: dotdash • 4: dash dot dot • 8: invisible. That is, the element is shown without a contour.

State variables

These are dynamic definitions of the availability of the element in online mode.

Invisibility	Specification of a boolean variable. If this returns TRUE, the element is invisible in online mode.
Input disabled	Specification of a boolean variable. If TRUE is returned, inputs for the element have no effect. Also, the element itself is greyed out in the visualization, to indicate that no user inputs are possible. If the visualization uses the user management [▶ 393], the elements for user groups with access right "only visible" are greyed out.

Input configuration

Here you can define the consequential action that should be performed when the user makes an input in the element in online mode. As long as no follow-on actions are defined, "Configure..." appears in the Properties field. Click on "Configure..." to open the [Input Configuration](#) [[▶ 406](#)], where you can assign follow-on actions. Each input action can be assigned any number of follow-on actions.

The following input events are available for an element:

- OnDialogClosed
- OnMouseClicked
- OnMouseDown
- OnMouseEnter
- OnMouseLeave
- OnMouseMove
- OnMouseUp

OnDialogClosed	This event is triggered if one of the dialog boxes that were opened for user inputs is closed within a visualization. Note: This property is not limited to the element, for which it is configured, but applies within the entire visualization. It therefore does not respond to each dialog closing action. Currently there is no way to define such a property for the entire visualization. It therefore has to be assigned to one of its elements.
OnMouseClicked	This mouse event is triggered when the cursor points to an element and a full mouse click (pressing and releasing the mouse button) is executed on this item.
OnMouseDown	This mouse event is triggered when the mouse button is pressed while the cursor points to an element. It is irrelevant where on the visualization the mouse button is released again.
OnMouseEnter	This mouse event is triggered when the cursor is dragged to the element. It is irrelevant whether the mouse button is pressed or released.
OnMouseLeave	This mouse event is triggered when the cursor leaves the element. It is irrelevant whether the mouse button is pressed or released.
OnMouseMove	This mouse event is triggered when the cursor is moved within the element. It is irrelevant whether the mouse button is pressed or released.
OnMouseUp	This mouse event occurs when the mouse button is released on the element. The mouse button was pressed prior to that outside of the element.

Input configuration - hotkeys

A hotkey can be used to define a key or key combination and link it with a follow-on action (e.g. MouseDown, MouseUp), which is to be executed when a key event occurs (KeyDown, KeyUp). By default the MouseDown action is executed on KeyDown (press key) and the MouseUp action on KeyUp (release key). This can be useful if a visualization is to be operated both via mouse actions and keyboard entries, since the input actions then only have to be configured once. The key configuration for an element is also managed in the [hotkeys configuration](#) [[▶ 392](#)] for the visualization. Any changes are always synchronized between this and the element properties editor.

Key	Assigning a key. A selection list contains all currently supported keys, e.g. M.
Event(s)	Definition of the event to be executed, if the key or key and modifier are used. Possible values that are available in a selection list: <ul style="list-style-type: none"> • No action • MouseDown action when the key is pressed • MouseUp action when the key is released • MouseDown/MouseUp action when the key is pressed/released
Shift	If this option is enabled, the key has to be used in combination with the Shift key.
Control	If this option is enabled, the key has to be used in combination with the Ctrl key.
Alt	If this option is enabled, the key has to be used in combination with the Alt key.

Input configuration – Keys

"Keys" can be used to specify that the value of a boolean project variable is set depending on the mouse behavior for the event "Keys".

Variable	Boolean variable whose value is TRUE if the mouse button is pressed while the cursor points to the element. The value becomes FALSE again when the mouse button is released or the cursor leaves the element.
FALSE keys	If this option is enabled, the key behavior described above is reversed for the corresponding variable. That is, when the mouse button pressed, the variable value is set to FALSE. When the button is, the variable value is set to TRUE.
Switch on enter if mouse trapped	If this option is enabled, the variable value behaves as described under 'Variables', as long as the cursor is within the element area. If the cursor leaves the element area while the mouse button is pressed, the variable is set to FALSE. However, it is automatically set to TRUE again, if the cursor returns to the element area without the mouse button being released. In other words, the system takes into account that the mouse is "trapped" as long as the mouse button is pressed, even if the cursor leaves the element area.

Input configuration - Switching

"Switching" can be used to specify that for this event the value of a boolean project variable is set depending on the mouse behavior.

Variable	Boolean variable whose value switches between TRUE and FALSE with each mouse click on the element. No switching occurs if the cursor is moved away from the element while the mouse button is pressed. Can be used to cancel a switching action.
Switch on release, if mouse trapped	If this option is enabled, the variable value behaves as described under 'Variables', as long as the cursor is within the element area. If the cursor leaves the element area while the mouse button is pressed and the mouse button is then released, the variable is switched anyway. In other words, the system takes into account that the mouse is "trapped" as long as the mouse button is pressed, even if the cursor leaves the element area.

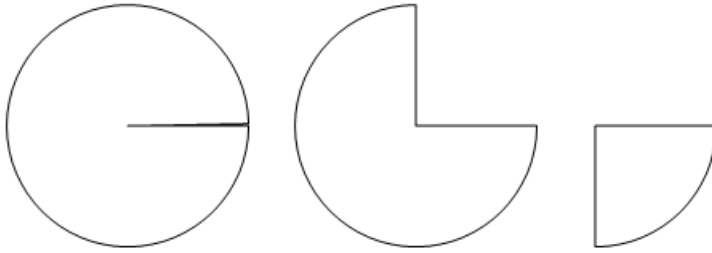
Access rights

This setting relates to the access rights for the individual element. Click to open the [Access rights dialog \[▶ 420\]](#). The setting is only available if a [user management \[▶ 393\]](#) was added to the PLC project. The following status messages are available:

Not set. All rights.	The default message is set, if the element is shown as available for all groups.
Rights are issued: Limited rights.	The message is set, if the element is shown with limited behavior for at least one group.

15.8.3.4 Circular sector

The element "Circular sector" basically represents a full circle with the central position at the point at which the element is placed on the visualization. The segment to be displayed can be adjusted in the [Settings \[▶ 498\]](#).




Properties editor

The properties of a visualization element - except alignment and order [[▶ 377](#)] - can all be configured in the properties editor [[▶ 385](#)]. By default, this editor opens next to the visualization editor, or it can be opened explicitly via the "Properties" command (which can be found in the View menu as standard).

A property can be modified by editing the field "Value". To this end, an input field, a selection list, a dialog or checkbox that can be activated is provided in this field, depending on the element type. The value field opens

- after a double-click,
- after a single click in a selected field,
- via the space bar, if the field was already selected.

If a variable is assigned,

- simply enter its name.
- Use the  button to open the input assistant for selecting a variable. The Variables category lists all variables that have already been defined in the project.



Working in the list of properties can be made easier with the aid of default, sorting and filter functions.

Element properties


All element properties and their descriptions are listed below.

Element name	The element name can be changed. Standard name is "GenElemInst_x". "x" stands for a sequential number.
Element type	The element type is entered here. For three element groups it is possible to switch between the corresponding elements by changing the element type: <ul style="list-style-type: none"> • rectangle, rounded rectangle and ellipse [▶ 475] • Polygon, polyline or Bézier curve [▶ 490] • dip switch, power switch, push switch, push switch with LED, rocker switch [▶ 532]

Additional settings

Start	Start of the circular sector line
End	End of the circular sector line
Variable for start	Variable that is to define the start of the circular sector line. Actuate the button  for help from the input assistant.
Variable for end	Variable that is to define the end of the circular sector line. Actuate the button  for help from the input assistant.
Only show circular arc	If this option is enabled, the circular sector is shown without radius line for start and end.

Center

When a value is edited, the corresponding element  is simultaneous moved in the [visualization editor](#) [[▶ 376](#)].

X	Horizontal position of the element pivot in pixels
Y	Vertical position of the element pivot in pixels

Colors

A color is defined based on a hexadecimal number consisting of red / green / blue (RGB) components. For each of these three colors, 256 (0-255) values are available, which can be entered here statically. The color can be selected from a selection list or via the color selection dialog, which can be opened via the button



. In addition, the level of transparency can be set for each color (0: fully transparent, 255: fully opaque).

Normal state • Frame color • Fill color	Select a frame and fill color for the default state. If the color switching variable is defined as FALSE, the element is in default state.
Alarm state • Frame color • Fill color	Select a frame and fill color for the element in alarm state. This is triggered, if the color switching variable is defined as TRUE.
Use color gradient	The checkbox is unticked by default. If it is ticked, the corresponding element is drawn with a color gradient.
Color gradient selection	The Gradient editor [▶ 405] opens.



Color gradient and transparency are not supported under Windows CE.

Appearance

The settings under Appearance are static definitions for the frame and the element filling.

Line width	Defines the frame line width in pixels. 0 codes the same as 1 and sets the line width to 1 pixel. If no frame is required, set the line type to invisible.
Fill type	Defines the fill type for the fill color: • Filled: The fill color is visible • Invisible: The fill color is invisible
Line type	Defines one of the following line types for the outline: • solid • dashed • dotted • dotdash • dash dot dot • Invisible: outline is invisible.



Texts

These properties are used for a static definition of the element labelling. Each can contain a [formatting sequence](#) [[▶ 617](#)], e.g. %s. In online mode the sequence is replaced by the contents of the variables defined in "Text variables".

Text	Enter a text. It is used to label the element. The text can contain a formatting sequence, e.g. %s. The corresponding variable is defined in "Text variables". Note: The text can also be entered directly. If the element is selected in the visualization editor, an input field can be opened by pressing the space bar.
Tooltip	Enter a text. It is used as a tooltip for the element and appears in the visualization only in online mode when the cursor is placed over an element. The text can contain a formatting sequence, e.g. %s. The corresponding variable is defined in "Text variables".




Text properties

These properties are used for a static definition of the font. A dynamic definition of the font is possible in the category "Font variables".

Horizontal alignment	Defines the horizontal alignment of the text through selection of: <ul style="list-style-type: none"> • Left • Centered • Right
Vertical alignment	Defines the vertical alignment of the text through selection of: <ul style="list-style-type: none"> • Top • Centered • Bottom
Text format	Defines the display of a text that is too long to be displayed completely in an element: <ul style="list-style-type: none"> • Default – the text extends beyond the element. • Line break – the text is automatically wrapped. • Ellipsis - the text is displayed as far as possible and then truncated with "...".
Font:	Defines the font through selection from predefined fonts: <ul style="list-style-type: none"> • Standard • Heading • Large heading • Title • Note <p>Press  to opens the dialog for user-defined font properties.</p>
Font color	Defines the font color for the element. Either from the selection list or via the dialog that opens when  is clicked.

Absolute movement

The element can be moved by changing the x- and y-position (pixels) of the top left corner of the element via an integer variable. Absolute coordinate values are used here.

<p>Motion</p> <ul style="list-style-type: none"> • X • Y 	<p>X: The integer variable entered here defines the current x-position of the top left corner of the element (in pixels). It can be used to move the element in x-direction. (A positive value moves the element from left to right).</p> <p>Y: The integer variable entered here defines the current y-position of the top left corner of the element (in pixels). It can be used to move the element in y-direction. (A positive value moves the element from top to bottom).</p>
<p>Rotation</p>	<p>The integer variable entered here defines the angle (angular degrees) for rotating the element around a rotation point.</p> <p>Positive values = clockwise</p> <p>Note: In contrast to the behavior with 'internal rotation' (see below), the element itself does not rotate. Click on the element to show the rotation point . It can be moved by pressing and holding the mouse button.</p>
<p>Scaling</p>	<p>The integer variable entered here defines the current scaling factor (percent). The element size is adjusted linearly according to this value. The value is implicitly divided by 1000, so that it is not necessary to use REAL variables in order to shrink the element. The scaling always refers to the rotation point (center). Click on the element to show the rotation point . It can be moved by pressing and holding the mouse button.</p>
<p>Internal rotation</p>	<p>The integer variable entered here defines the angle (angular degrees) by which the element is rotated around its rotation point; positive values=mathematically positive=clockwise. In contrast to "Rotation" (see above), the element itself rotates. Click on the element to show the rotation point (center) . It can be moved by pressing and holding the mouse button.</p>

Dynamic texts

These parameters are used to define dynamic texts originating from [text lists \[▶ 138\]](#). This enables [language change \[▶ 616\]](#), for example.

A further possibility to define a text dynamically is to supply the text via a string variable. (see the category "Text variables")

<p>Text list</p>	<p>Name of the text list, as used in the project tree, as a string</p> <p>Example: 'TL_ErrorList'</p>
<p>Textindex</p>	<p>Index (ID) of the text, as defined in the text list, as a string. It can be specified directly in a static manner or as a string variable.</p>
<p>Tooltipindex</p>	<p>Index (ID) of the tooltip text, as defined in the text list, as a string. It can be specified directly in a static manner or as a string variable.</p>

Text variables

You can display text that is stored in a variable. To do this, first add a formatting sequence in the text that is defined under the "Texts" property. Then assign a variable. In online mode the [formatting sequence \[▶ 617\]](#) is replaced by the contents of the variables.

Example %f:

Enter "Result: %2.5f" in the "Texts" property. Enter "fValue" in the "Text variables" property. The variable must be a defined IEC variable. The element will then be labeled in online mode with "Result: 12.12345" if fValue = 12.1234567.

<p>Text variable</p>	<p>Variable (of default data type) containing the information to be displayed. The type must match the formatting sequence in the "Texts" setting.</p>
<p>Tooltipvariable</p>	<p>Variable of type String containing the tooltip text to be displayed. The entry in the property "Texts" must contain a formatting sequence.</p>

Font variables

These variables are used for dynamic font definitions for element texts via project variables. Static definitions are configured under "Text properties".

Font name	Specification of a variable of the type String that contains the font name that is to be used to label the element. (name specified as in the standard font dialog) Example: MAIN.sFont (sFont := 'Arial');
Size	Variable of type INT containing the size of the element text in pixels, as in the default dialog 'Font'. Example: MAIN.nHeight (nHeight := 16;)
Flags	Variable of type DWORD for defining the font display via one of the flag values listed below. A combined definition can be achieved by adding the respective flag values and specifying the sum. <ul style="list-style-type: none"> • 1: italics • 2: bold • 4: underlined • 8: deleted Example: MAIN.nFlag (nFlag := 6;) The text is shown bold and underlined.
Character set	The character set to be used for the font can be defined via the default font number. This number can be specified via a DWORD variable (see also the definition in the default font dialog)
Color	Variable of type DWORD for defining the color of the element text.
Flags for the text alignment	Variable of type DWORD for defining the text alignment via one of the flag values listed below. A combined definition can be achieved by adding the respective flag values and specifying the sum. <ul style="list-style-type: none"> • 0: top left • 1: centered horizontally • 2: right • 4: centered vertically • 8: below Example MAIN.nFlag (nFlag := 5;) The text is displayed centered horizontally and vertically.

Color variables

The color variables used for dynamic definition of the element colors via project variables of type DWORD. A color is defined based on a hexadecimal number consisting of red, green and blue (RGB) components. In addition, the variables are used to specify the transparency of the color (FF: fully opaque - 00: fully transparent). The DWORD has the following structure: 16#TTRRGGBB



The structure of the hexadecimal number differs, compared to TwinCAT 2. In TwinCAT 3, in addition to the RGB components, the color transparency can also be defined with the hexadecimal number. The transparency is indicated by the first two digits after "16#". Colors with definitions starting with "16#00" are invisible, since they are fully transparent.

Example:

```
nFillColor := 16#FF8FE03F;
```

- FF: transparency (fully opaque)
- 8F: red
- E0: green

- 3F: blue

Color change	Boolean variable, which controls the switching of the element color between "normal state" (variable = FALSE) and "alarm state" (variable = TRUE).
Normal state • Frame color • Fill color	Variables of type DWORD for defining the frame and fill color for the element. They overwrite the values that are currently defined in "Colors". The values in the project variable are used, if the variable defined in "Color change" is FALSE.
Alarm state • Frame color • Fill color	Variables of type DWORD for defining the frame and fill color for the element in alarm state. They overwrite the values that are currently defined in "Colors". The values in the project variable are used, if the variable defined in "Color change" is TRUE.



Transparency is not supported under Windows CE.

State variables

These are dynamic definitions of the availability of the element in online mode.

Invisibility	Specification of a boolean variable. If this returns TRUE, the element is invisible in online mode.
Input disabled	Specification of a boolean variable. If TRUE is returned, inputs for the element have no effect. Also, the element itself is greyed out in the visualization, to indicate that no user inputs are possible. If the visualization uses the <u>user management</u> [▶ 393], the elements for user groups with access right "only visible" are greyed out.

Appearance variables

To be used for a dynamic definition of the contour appearance and the element filling. Static definitions are specified under "Appearance".

Line width	Variable of type integer for defining the line width of the element in pixels. It overwrites the fixed value that was specified under "Appearance".
Fill type	Variable of type DWORD for defining the element filling. The color specified under color variables can be displayed or ignored: • Variable value = 0: filled • Variable value > 0: invisible, i.e. no filling is visible
Line type	Variable of type DWORD for defining the contour. The following values correspond to the following line type: • 0: solid • 1: dashed • 2: dotted • 3: dotdash • 4: dash dot dot • 8: invisible. That is, the element is shown without a contour.

Input configuration

Here you can define the consequential action that should be performed when the user makes an input in the element in online mode. As long as no follow-on actions are defined, "Configure..." appears in the Properties field. Click on "Configure..." to open the Input Configuration [[▶ 406](#)], where you can assign follow-on actions. Each input action can be assigned any number of follow-on actions.

The following input events are available for an element:

- OnDialogClosed

- OnMouseClicked
- OnMouseDown
- OnMouseEnter
- OnMouseLeave
- OnMouseMove
- OnMouseUp

OnDialogClosed	This event is triggered if one of the dialog boxes that were opened for user inputs is closed within a visualization. Note: This property is not limited to the element, for which it is configured, but applies within the entire visualization. It therefore does not respond to each dialog closing action. Currently there is no way to define such a property for the entire visualization. It therefore has to be assigned to one of its elements.
OnMouseClicked	This mouse event is triggered when the cursor points to an element and a full mouse click (pressing and releasing the mouse button) is executed on this item.
OnMouseDown	This mouse event is triggered when the mouse button is pressed while the cursor points to an element. It is irrelevant where on the visualization the mouse button is released again.
OnMouseEnter	This mouse event is triggered when the cursor is dragged to the element. It is irrelevant whether the mouse button is pressed or released.
OnMouseLeave	This mouse event is triggered when the cursor leaves the element. It is irrelevant whether the mouse button is pressed or released.
OnMouseMove	This mouse event is triggered when the cursor is moved within the element. It is irrelevant whether the mouse button is pressed or released.
OnMouseUp	This mouse event occurs when the mouse button is released on the element. The mouse button was pressed prior to that outside of the element.

Input configuration - hotkeys

A hotkey can be used to define a key or key combination and link it with a follow-on action (e.g. MouseDown, MouseUp), which is to be executed when a key event occurs (KeyDown, KeyUp). By default the MouseDown action is executed on KeyDown (press key) and the MouseUp action on KeyUp (release key). This can be useful if a visualization is to be operated both via mouse actions and keyboard entries, since the input actions then only have to be configured once. The key configuration for an element is also managed in the [hotkeys configuration \[▶ 392\]](#) for the visualization. Any changes are always synchronized between this and the element properties editor.

Key	Assigning a key. A selection list contains all currently supported keys, e.g. M.
Event(s)	Definition of the event to be executed, if the key or key and modifier are used. Possible values that are available in a selection list: <ul style="list-style-type: none"> • No action • MouseDown action when the key is pressed • MouseUp action when the key is released • MouseDown/MouseUp action when the key is pressed/released
Shift	If this option is enabled, the key has to be used in combination with the Shift key.
Control	If this option is enabled, the key has to be used in combination with the Ctrl key.
Alt	If this option is enabled, the key has to be used in combination with the Alt key.

Input configuration – Keys

"Keys" can be used to specify that the value of a boolean project variable is set depending on the mouse behavior for the event "Keys".

Variable	Boolean variable whose value is TRUE if the mouse button is pressed while the cursor points to the element. The value becomes FALSE again when the mouse button is released or the cursor leaves the element.
FALSE keys	If this option is enabled, the key behavior described above is reversed for the corresponding variable. That is, when the mouse button pressed, the variable value is set to FALSE. When the button is, the variable value is set to TRUE.
Switch on enter if mouse trapped	If this option is enabled, the variable value behaves as described under 'Variables', as long as the cursor is within the element area. If the cursor leaves the element area while the mouse button is pressed, the variable is set to FALSE. However, it is automatically set to TRUE again, if the cursor returns to the element area without the mouse button being released. In other words, the system takes into account that the mouse is "trapped" as long as the mouse button is pressed, even if the cursor leaves the element area.

Input configuration - Switching

"Switching" can be used to specify that for this event the value of a boolean project variable is set depending on the mouse behavior.

Variable	Boolean variable whose value switches between TRUE and FALSE with each mouse click on the element. No switching occurs if the cursor is moved away from the element while the mouse button is pressed. Can be used to cancel a switching action.
Switch on release, if mouse trapped	If this option is enabled, the variable value behaves as described under 'Variables', as long as the cursor is within the element area. If the cursor leaves the element area while the mouse button is pressed and the mouse button is then released, the variable is switched anyway. In other words, the system takes into account that the mouse is "trapped" as long as the mouse button is pressed, even if the cursor leaves the element area.

Access rights

This setting relates to the access rights for the individual element. Click to open the [Access rights dialog \[▶ 420\]](#). The setting is only available if a [user management \[▶ 393\]](#) was added to the PLC project. The following status messages are available:

Not set. All rights.	The default message is set, if the element is shown as available for all groups.
Rights are issued: Limited rights.	The message is set, if the element is shown with limited behavior for at least one group.

15.8.3.5 Image

This element can be used to add an image to the visualization. An image is allocated to the image element by specifying the corresponding static ID in combination with the name of the [image pool \[▶ 146\]](#). The setting "[Image ID variable \[▶ 509\]](#)" can be used to define the image to be displayed dynamically.



A background image for the visualization page can be set via a [background dialog \[▶ 378\]](#) in the visualization editor.


Properties editor

The properties of a visualization element - except [alignment and order \[▶ 377\]](#) - can all be configured in the [properties editor \[▶ 385\]](#). By default, this editor opens next to the visualization editor, or it can be opened explicitly via the "Properties" command (which can be found in the View menu as standard).

A property can be modified by editing the field "Value". To this end, an input field, a selection list, a dialog or checkbox that can be activated is provided in this field, depending on the element type. The value field opens



- after a double-click,
- after a single click in a selected field,
- via the space bar, if the field was already selected.

If a variable is assigned,

- simply enter its name.
- Use the  button to open the input assistant for selecting a variable. The Variables category lists all variables that have already been defined in the project.

Working in the list of properties can be made easier with the aid of default, sorting and filter functions.

Additional settings


<p>Static ID</p>	<p>ID of the image file for a static definition.</p> <p>Enter the ID of the image file, as defined in the corresponding image pool (string). The name of the image pool should be prefixed in order to make the entry unambiguous. (This is not necessary if the image file is managed in the global image pool, as this pool is always scanned first.)</p> <p>Example: IP_ImagePool.ButtonImage</p> <p>Click the  button to open the "input assistant [▶ 515]" dialog with a list of all available image pools and associated images.</p>
<p>Draw frame</p>	<p>If this option is enabled, the image file is shown with a frame.</p>
<p>Truncate</p>	<p>If this option is enabled together with scaling type "Unscaled", only the part of the image that fits in the element is shown.</p>
<p>Transparency</p>	<p>If this option is enabled, the color specified in the property "Transparency color" is shown transparent.</p>
<p>Transparency color</p>	<p>The button  opens the color selection dialog for selecting a color to be displayed transparent if the option "Transparency" is enabled.</p>
<p>Scaling type</p>	<p>Here you can define how the image file responds to changes in the element frame size:</p> <ul style="list-style-type: none"> • Isotropic <p>The image retains its proportions. That is, the height/width ratio is retained, even if the height and width of the element frame are changed separately.</p> <ul style="list-style-type: none"> • Anisotropic <p>The image adapts itself to the size of the element frame, i.e. the height and width can be changed independently of each other.</p> <ul style="list-style-type: none"> • Unscaled <p>The image retains its original size, even if the size of the element frame changes. Another factor to consider is whether the option "Truncate" is enabled.</p> <p>With this setting the element size is automatically adjusted to the image size whenever a new image ID is allocated.</p>
<p>Horizontal alignment</p>	<p>This setting is only available if the option "Expert" in the properties editor [▶ 385] is enabled and if the image scaling type is "Isotropic". It is used to define the horizontal alignment relative to the element frame.</p> <ul style="list-style-type: none"> • Left • Centered • Right
<p>Vertical alignment</p>	<p>This setting is only available if the option "Expert" in the properties editor [▶ 385] is enabled and if the image scaling type is "Isotropic". It is used to define the vertical alignment relative to the element frame.</p> <ul style="list-style-type: none"> • Top • Centered • Bottom

Position

Here you can define the position (X/Y coordinates) and size (width and height) of the element in pixels. The origin is in the top left corner of the window. The positive x-axis is on the right, the positive y-axis runs downwards. If the values edited, the displayed element is simultaneously modified in the [visualization editor \[▶ 376\]](#).

X	Horizontal position in pixels – X=0 is the left edge of the window.
Y	Vertical position in pixels – Y=0 is the upper edge of the window.
Width	Width of the element in pixels
Height	Height of the element in pixels

Center

When a value is edited, the corresponding element  is simultaneous moved in the [visualization editor](#) [[▶ 376](#)].

X	Horizontal position of the element pivot in pixels
Y	Vertical position of the element pivot in pixels

Colors

A color is defined based on a hexadecimal number consisting of red / green / blue (RGB) components. For each of these three colors, 256 (0-255) values are available, which can be entered here statically. The color can be selected from a selection list or via the color selection dialog, which can be opened via the button



. In addition, the level of transparency can be set for each color (0: fully transparent, 255: fully opaque).

Color	Select a frame and fill color for the default state. If the color switching variable is defined as FALSE, the element is in default state.
Alarm color	Select a frame and fill color for the element in alarm state. This is triggered, if the color switching variable is defined as TRUE.



Transparency is not supported under Windows CE.

Appearance

Line width	Defines the frame line width in pixels. 0 codes the same as 1 and sets the line width to 1 pixel. If no frame is required, set the line type to invisible.
Line type	Defines one of the following line types for the outline: <ul style="list-style-type: none"> • solid • dashed • dotted • dotdash • dash dot dot • Invisible: outline is invisible.

Texts

These properties are used for a static definition of the element labelling. Each can contain a [formatting sequence](#) [[▶ 617](#)], e.g. %s. In online mode the sequence is replaced by the contents of the variables defined in "Text variables".

Text	Enter a text. It is used to label the element. The text can contain a formatting sequence, e.g. %s. The corresponding variable is defined in "Text variables". Note: The text can also be entered directly. If the element is selected in the visualization editor, an input field can be opened by pressing the space bar.
Tooltip	Enter a text. It is used as a tooltip for the element and appears in the visualization only in online mode when the cursor is placed over an element. The text can contain a formatting sequence, e.g. %s. The corresponding variable is defined in "Text variables".

Text properties

These properties are used for a static definition of the font. A dynamic definition of the font is possible in the category "Font variables".



Horizontal alignment	Defines the horizontal alignment of the text through selection of: <ul style="list-style-type: none"> • Left • Centered • Right
Vertical alignment	Defines the vertical alignment of the text through selection of: <ul style="list-style-type: none"> • Top • Centered • Bottom
Text format	Defines the display of a text that is too long to be displayed completely in an element: <ul style="list-style-type: none"> • Default – the text extends beyond the element. • Line break – the text is automatically wrapped. • Ellipsis - the text is displayed as far as possible and then truncated with "...".
Font:	Defines the font through selection from predefined fonts: <ul style="list-style-type: none"> • Standard • Heading • Large heading • Title • Note <p>Press  to opens the dialog for user-defined font properties.</p>
Font color	Defines the font color for the element. Either from the selection list or via the dialog that opens when  is clicked.

Image ID variable

Image ID	Project variable of the type String that contains the image ID with which the element is to be displayed. The ID is defined in the image pool [► 146] . The name of the image pool should be prefixed in order to make the entry unambiguous. For image files that are managed in the "GlobalImagePool", it is not necessary to specify an image pool, since this image pool is always searched first.
----------	--

Dynamic image

With this property, an image file can be reloaded at runtime. This allows camera images to be refreshed, for example.




The image file being reloaded must

- be exchanged in the client directory of the device that the PLC HMI client has been started on (C:\TwinCAT\3.1\Components\Plc\Tc3PlcHmi\Port_X\Visu) for PLC HMIs with a build <4022.28.
- be exchanged in the boot directory of the device that the control code is executed on (C:\TwinCAT\3.1\Boot\Plc\Port_X\Visu) for PLC HMIs with a build >=4022.28 and the PLC HMI Web.
- be exchanged in the boot directory of the device that the control code is executed on (: \ProgramData\Beckhoff\TwinCAT\3.1\Boot\Plc\Port_X\Visu) for PLC HMIs and PLC HMI Web with a build >=TC3.1.4026.0.

Bitmap version	Variable of an integer data type that contains the image version. If the value of the variable changes, the visualization reloads the image referenced in the "Image ID" property and displays it.
----------------	--

Absolute movement

The element can be moved by changing the x- and y-position (pixels) of the top left corner of the element via an integer variable. Absolute coordinate values are used here.

<p>Motion</p> <ul style="list-style-type: none"> • X • Y 	<p>X: The integer variable entered here defines the current x-position of the top left corner of the element (in pixels). It can be used to move the element in x-direction. (A positive value moves the element from left to right).</p> <p>Y: The integer variable entered here defines the current y-position of the top left corner of the element (in pixels). It can be used to move the element in y-direction. (A positive value moves the element from top to bottom).</p>
<p>Rotation</p>	<p>The integer variable entered here defines the angle (angular degrees) for rotating the element around a rotation point.</p> <p>Positive values = clockwise</p> <p>Note: In contrast to the behavior with 'internal rotation' (see below), the element itself does not rotate. Click on the element to show the rotation point . It can be moved by pressing and holding the mouse button.</p>
<p>Scaling</p>	<p>The integer variable entered here defines the current scaling factor (percent). The element size is adjusted linearly according to this value. The value is implicitly divides by 1000, so that it is not necessary to use REAL variables in order to shrink the element. The scaling always refers to the rotation point (center). Click on the element to show the rotation point . It can be moved by pressing and holding the mouse button.</p>
<p>Internal rotation</p>	<p>The integer variable entered here defines the angle (angular degrees) by which the element is rotated around its rotation point; positive values=mathematically positive=clockwise. In contrast to "Rotation" (see above), the element itself rotates. Click on the element to show the rotation point (center) . It can be moved by pressing and holding the mouse button.</p>

Relative movement

The element can be moved relative to its fixed position. The top left and bottom right edges of the element are moved in x- or y-direction by a value (pixels) defined by an integer variable. In contrast to an absolute movement, a relative position is defined, i.e. the distance to the original position. This function can be used to change the shape of the element. Positive values move the horizontal edges downwards and/or the vertical edges to the right.

<p>Top left movement</p> <ul style="list-style-type: none"> • X • Y 	<ul style="list-style-type: none"> • X: Integer variable, the value of which indicates the number of pixels by which the top left corner is moved in x-direction. • Y: Integer variable, the value of which indicates the number of pixels by which the top left corner is moved in y-direction.
<p>Bottom right movement</p> <ul style="list-style-type: none"> • X • Y 	<ul style="list-style-type: none"> • X: Integer variable, the value of which indicates the number of pixels by which the bottom right corner is moved in x-direction. • Y: Integer variable, the value of which indicates the number of pixels by which the bottom right corner is moved in y-direction.

Text variables

You can display text that is stored in a variable. To do this, first add a formatting sequence in the text that is defined under the "Texts" property. Then assign a variable. In online mode the formatting sequence [▶ 617](#) is replaced by the contents of the variables.

Example %f:

Enter "Result: %2.5f" in the "Texts" property. Enter "fValue" in the "Text variables" property. The variable must be a defined IEC variable. The element will then be labeled in online mode with "Result: 12.12345" if fValue = 12.1234567.

Text variable	Variable (of default data type) containing the information to be displayed. The type must match the formatting sequence in the "Texts" setting.
Tooltiptvariable	Variable of type String containing the tooltip text to be displayed. The entry in the property "Texts" must contain a formatting sequence.

Dynamic texts

These parameters are used to define dynamic texts originating from [text lists \[▶ 138\]](#). This enables [language change \[▶ 616\]](#), for example.

A further possibility to define a text dynamically is to supply the text via a string variable. (see the category "Text variables")

Text list	Name of the text list, as used in the project tree, as a string Example: 'TL_ErrorList'
Textindex	Index (ID) of the text, as defined in the text list, as a string. It can be specified directly in a static manner or as a string variable.
Tooltiptindex	Index (ID) of the tooltip text, as defined in the text list, as a string. It can be specified directly in a static manner or as a string variable.

Font variables

These variables are used for dynamic font definitions for element texts via project variables. Static definitions are configured under "Text properties".

Font name	Specification of a variable of the type String that contains the font name that is to be used to label the element. (name specified as in the standard font dialog) Example: MAIN.sFont (sFont := 'Arial');
Size	Variable of type INT containing the size of the element text in pixels, as in the default dialog 'Font'. Example: MAIN.nHeight (nHeight := 16;)
Flags	Variable of type DWORD for defining the font display via one of the flag values listed below. A combined definition can be achieved by adding the respective flag values and specifying the sum. <ul style="list-style-type: none"> • 1: italics • 2: bold • 4: underlined • 8: deleted Example: MAIN.nFlag (nFlag := 6;) The text is shown bold and underlined.
Character set	The character set to be used for the font can be defined via the default font number. This number can be specified via a DWORD variable (see also the definition in the default font dialog)
Color	Variable of type DWORD for defining the color of the element text.
Flags for the text alignment	Variable of type DWORD for defining the text alignment via one of the flag values listed below. A combined definition can be achieved by adding the respective flag values and specifying the sum. <ul style="list-style-type: none"> • 0: top left • 1: centered horizontally • 2: right • 4: centered vertically • 8: below Example MAIN.nFlag (nFlag := 5;) The text is displayed centered horizontally and vertically.

State variables

These are dynamic definitions of the availability of the element in online mode.

Invisibility	Specification of a boolean variable. If this returns TRUE, the element is invisible in online mode.
Input disabled	Specification of a boolean variable. If TRUE is returned, inputs for the element have no effect. Also, the element itself is greyed out in the visualization, to indicate that no user inputs are possible. If the visualization uses the user management [► 393], the elements for user groups with access right "only visible" are grayed out.

Input configuration

Here you can define the consequential action that should be performed when the user makes an input in the element in online mode. As long as no follow-on actions are defined, "Configure..." appears in the Properties field. Click on "Configure..." to open the [Input Configuration](#) [► 406], where you can assign follow-on actions. Each input action can be assigned any number of follow-on actions.

The following input events are available for an element:

- OnDialogClosed

- OnMouseClicked
- OnMouseDown
- OnMouseEnter
- OnMouseLeave
- OnMouseMove
- OnMouseUp

OnDialogClosed	This event is triggered if one of the dialog boxes that were opened for user inputs is closed within a visualization. Note: This property is not limited to the element, for which it is configured, but applies within the entire visualization. It therefore does not respond to each dialog closing action. Currently there is no way to define such a property for the entire visualization. It therefore has to be assigned to one of its elements.
OnMouseClicked	This mouse event is triggered when the cursor points to an element and a full mouse click (pressing and releasing the mouse button) is executed on this item.
OnMouseDown	This mouse event is triggered when the mouse button is pressed while the cursor points to an element. It is irrelevant where on the visualization the mouse button is released again.
OnMouseEnter	This mouse event is triggered when the cursor is dragged to the element. It is irrelevant whether the mouse button is pressed or released.
OnMouseLeave	This mouse event is triggered when the cursor leaves the element. It is irrelevant whether the mouse button is pressed or released.
OnMouseMove	This mouse event is triggered when the cursor is moved within the element. It is irrelevant whether the mouse button is pressed or released.
OnMouseUp	This mouse event occurs when the mouse button is released on the element. The mouse button was pressed prior to that outside of the element.

Input configuration - hotkeys

A hotkey can be used to define a key or key combination and link it with a follow-on action (e.g. MouseDown, MouseUp), which is to be executed when a key event occurs (KeyDown, KeyUp). By default the MouseDown action is executed on KeyDown (press key) and the MouseUp action on KeyUp (release key). This can be useful if a visualization is to be operated both via mouse actions and keyboard entries, since the input actions then only have to be configured once. The key configuration for an element is also managed in the [hotkeys configuration \[▶ 392\]](#) for the visualization. Any changes are always synchronized between this and the element properties editor.

Key	Assigning a key. A selection list contains all currently supported keys, e.g. M.
Event(s)	Definition of the event to be executed, if the key or key and modifier are used. Possible values that are available in a selection list: <ul style="list-style-type: none"> • No action • MouseDown action when the key is pressed • MouseUp action when the key is released • MouseDown/MouseUp action when the key is pressed/released
Shift	If this option is enabled, the key has to be used in combination with the Shift key.
Control	If this option is enabled, the key has to be used in combination with the Ctrl key.
Alt	If this option is enabled, the key has to be used in combination with the Alt key.

Element properties

All element properties and their descriptions are listed below.

Element name	The element name can be changed. Standard name is "GenElemInst_x". "x" stands for a sequential number.
Element type	The element type is entered here. For three element groups it is possible to switch between the corresponding elements by changing the element type: <ul style="list-style-type: none"> • rectangle, rounded rectangle and ellipse [► 475] • Polygon, polyline or Bézier curve [► 490] • dip switch, power switch, push switch, push switch with LED, rocker switch [► 532]

Color variables

The color variables used for dynamic definition of the element colors via project variables of type DWORD. A color is defined based on a hexadecimal number consisting of red, green and blue (RGB) components. In addition, the variables are used to specify the transparency of the color (FF: fully opaque - 00: fully transparent). The DWORD has the following structure: 16#TTRRGGBB

i The structure of the hexadecimal number differs from TwinCAT 2. In TwinCAT 3 it is possible to define the transparency of the color with the hexadecimal number in addition to the RGB proportions. The transparency is indicated by the first two digits after "16#". Colors with definitions starting with "16#00" are invisible, since they are fully transparent.

Example:

nFillColor := 16#FF8FE03F;

- FF: transparency (fully opaque)
- 8F: red
- E0: green
- 3F: blue

Color change	Boolean variable, which controls the switching of the element color between "normal state" (variable = FALSE) and "alarm state" (variable = TRUE).
Normal state	Variable of type DWORD for defining the element color. It overwrites the value that is currently defined in "Colors". The value in the project variable is used, if the variable defined in "Color change" is FALSE.
Alarm state	Variable of type DWORD for defining the element color in alarm state. It overwrites the value that is currently defined in "Colors". The value in the project variable is used, if the variable defined in "Color change" is TRUE.


i Transparency is not supported under Windows CE.

Access rights

This setting relates to the access rights for the individual element. Click to open the [Access rights dialog \[► 420\]](#). The setting is only available if a [user management \[► 393\]](#) was added to the PLC project. The following status messages are available:

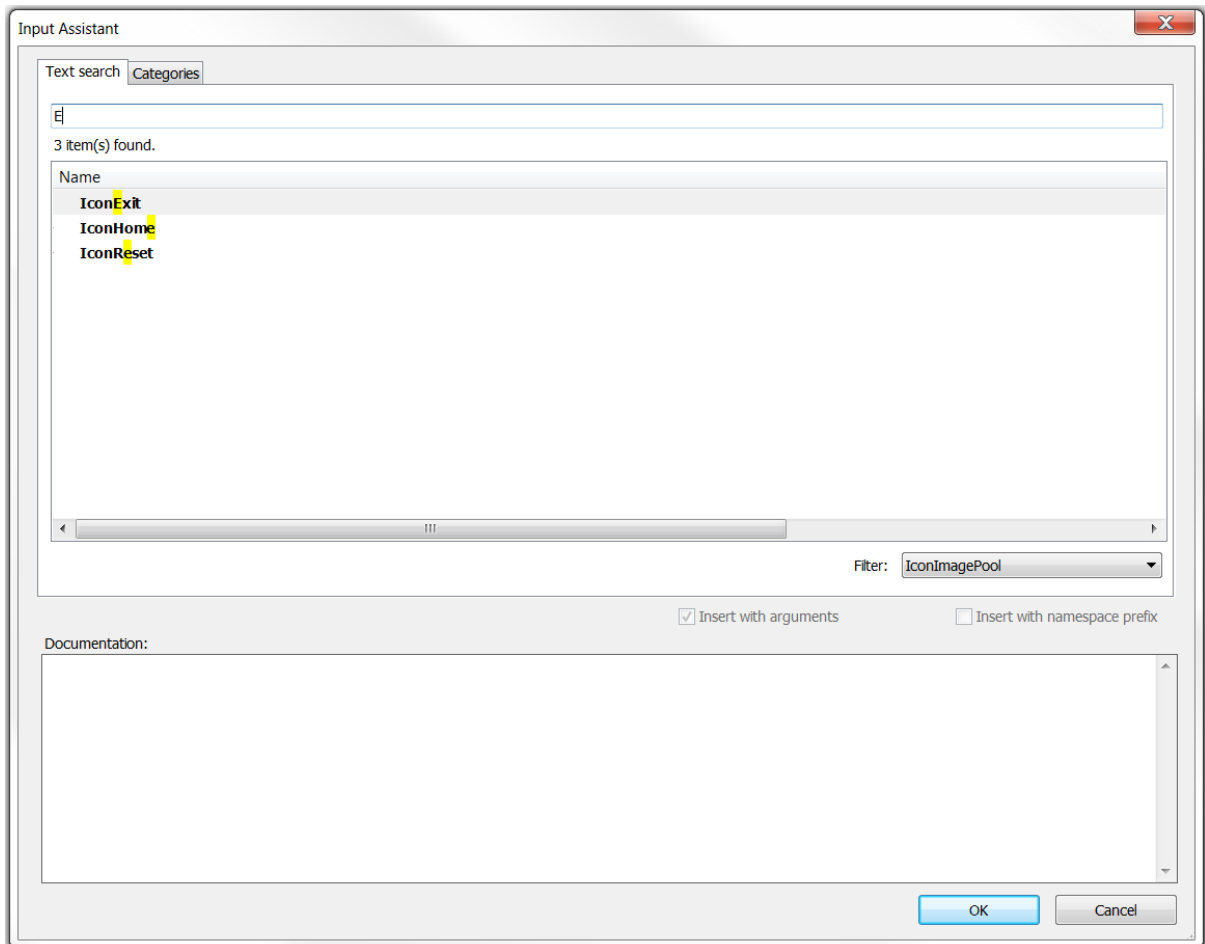
Not set. All rights.	The default message is set, if the element is shown as available for all groups.
Rights are issued: Limited rights.	The message is set, if the element is shown with limited behavior for at least one group.

15.8.3.5.1 Input Assistant

The input assistant for images from [image pools](#) [▶ 146] opens automatically when a visualization element of type [image](#) [▶ 505] is added to a visualization page for the first time. This dialog can also be reached by selecting the value field "[Static ID](#) [▶ 507]" in the properties of an image element and then clicking the  button.

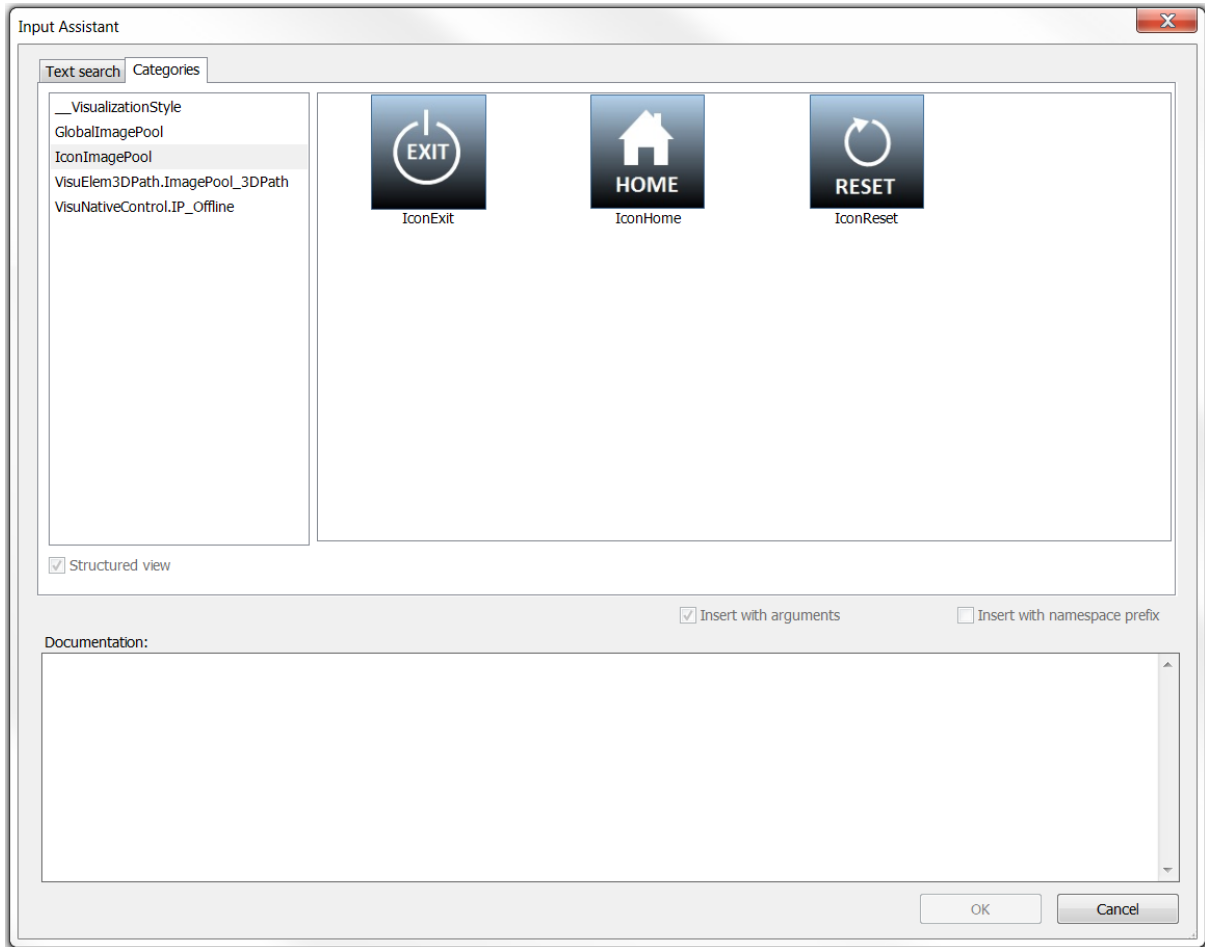
This dialog offers two options for selecting an image from the image pools that are present in the PLC project:

- Text search



Enter the name or part of the name of the required image file in the line editor at the top. All image file names that match the text are displayed. In addition, you can set a filter to narrow the search to a particular image pool.

- Categories



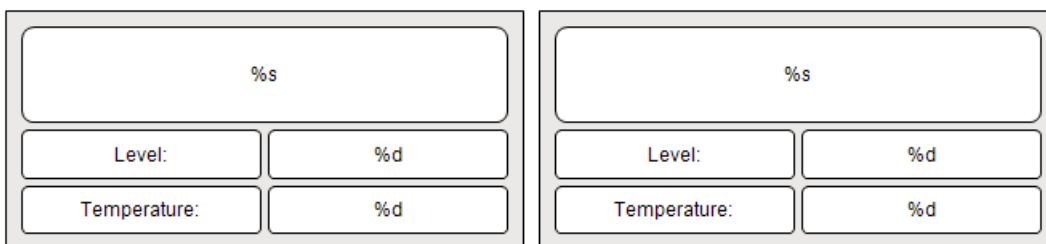
All image pools available in the project are listed in the column on the left. When an image pool is selected, a preview of the images is displayed on the right.

Use one of the two option to select an image, then close the dialog with OK. Alternatively, you can double-click on the required image. Once the dialog has been closed, the image is fully referenced in the property "Static ID [▶ 507]" with <name of image pool>.<name of image>.

15.8.3.6 Frame

A frame element is used

- As a reference to another visualization [▶ 612]
- As an interface for placeholders [▶ 612]
- For switching between different visualization pages within a frame [▶ 612]




Properties editor

The properties of a visualization element - except alignment and order [▶ 377] - can all be configured in the properties editor [▶ 385]. By default, this editor opens next to the visualization editor, or it can be opened explicitly via the "Properties" command (which can be found in the View menu as standard).

A property can be modified by editing the field "Value". To this end, an input field, a selection list, a dialog or checkbox that can be activated is provided in this field, depending on the element type. The value field opens

- after a double-click,
- after a single click in a selected field,
- via the space bar, if the field was already selected.

If a variable is assigned,

- simply enter its name.
- Use the  button to open the input assistant for selecting a variable. The Variables category lists all variables that have already been defined in the project.

Working in the list of properties can be made easier with the aid of default, sorting and filter functions.

Element properties

All element properties and their descriptions are listed below.

Element name	The element name can be changed. Standard name is "GenElemInst_x". "x" stands for a sequential number.
Element type	The element type is entered here. For three element groups it is possible to switch between the corresponding elements by changing the element type: <ul style="list-style-type: none"> • rectangle, rounded rectangle and ellipse [► 475] • Polygon, polyline or Bézier curve [► 490] • dip switch, power switch, push switch, push switch with LED, rocker switch [► 532]

Additional settings

Truncate	If this option is enabled together with scaling type "Unscaled", only the part of the visualization that fits in the frame is shown.
Draw frame	If this option is enabled, the referenced visualization is shown with a frame.
Scaling type	Here you can specify how the frame responds to changes in the size of the visualization: <ul style="list-style-type: none"> • Isotropic The frame retains its proportions. The ratio of height and width of the visualization can be changed independently. • Anisotropic The frame is based on the size of the visualization, so that height and width of the referencing visualization can be changed independently. • Unscaled The original size of the frame is retained, irrespective of the visualization size. If the option 'Truncate' is also enabled, only the part that fits is displayed. • Unscaled and scrollable If this option is used, the referenced visualization is shown without scaling. If it is larger than the window area of the frame, the frame is provided with scrollbars, so that the displayed area of the visualization can be moved. To set the position of the scrollbar with a variable, use the properties "Variable scroll position horizontal" or "Variable scroll position vertical".
Deactivate background drawing	To optimize the performance of the visualization, the non-animated elements of the frame element are drawn as a background bitmap. This could result in elements being shown not in the expected order. The function can be disabled to avoid this behavior.

Scrollbar settings

The scrollbar settings are only visible, if the scaling type "Unscaled and scrollable" is entered. It is highly recommended to use the variables on a client-specific basis. In this case, if the variables change, or if a scrollbar is moved with the mouse, the change only affects the frame of the respective client. Otherwise all clients are updated.

Variable scroll position horizontal	Variable containing the position of the horizontal scrollbar.
Variable scroll position vertical	Variable containing the position of the vertical scrollbar.

Client-specific variable for the scroll position:

```
PROGRAM MAIN
VAR
    aScrollPositionsHorizontal : ARRAY[0..20] OF INT;
    aScrollPositionsVertical : ARRAY[0..20] OF INT;
END_VAR
```

Specified properties for the frame element:

Variable scroll position horizontal	MAIN.aScrollPositionsHorizontal[CURRENTCLIENTID]
Variable scroll position vertical	MAIN.aScrollPositionsVertical[CURRENTCLIENTID]

Referenced visualizations


Here you can open the dialog "Frame selection [▶ 525]", which can be used to select the visualization pages to be referenced. Once one or several visualization pages have been selected, they are listed below with their placeholders [▶ 612], if available. If the placeholders change, the dialog "Updating the frame parameters [▶ 526]" automatically opens for all instances.

Position

Here you can define the position (X/Y coordinates) and size (width and height) of the element in pixels. The origin is in the top left corner of the window. The positive x-axis is on the right, the positive y-axis runs downwards. If the values edited, the displayed element is simultaneously modified in the visualization editor [▶ 376].

X	Horizontal position in pixels – X=0 is the left edge of the window.
Y	Vertical position in pixels – Y=0 is the upper edge of the window.
Width	Width of the element in pixels
Height	Height of the element in pixels

Center

When a value is edited, the corresponding element  is simultaneous moved in the visualization editor [▶ 376].

X	Horizontal position of the element pivot in pixels
Y	Vertical position of the element pivot in pixels

Colors

A color is defined based on a hexadecimal number consisting of red / green / blue (RGB) components. For each of these three colors, 256 (0-255) values are available, which can be entered here statically. The color can be selected from a selection list or via the color selection dialog, which can be opened via the button



. In addition, the level of transparency can be set for each color (0: fully transparent, 255: fully opaque).

Color	Select a frame and fill color for the default state. If the color switching variable is defined as FALSE, the element is in default state.
Alarm color	Select a frame and fill color for the element in alarm state. This is triggered, if the color switching variable is defined as TRUE.



Transparency is not supported under Windows CE.

Appearance

Line width	Defines the frame line width in pixels. 0 codes the same as 1 and sets the line width to 1 pixel. If no frame is required, set the line type to invisible.
Line type	Defines one of the following line types for the outline: <ul style="list-style-type: none"> • solid • dashed • dotted • dotdash • dash dot dot • Invisible: outline is invisible.



Texts

These properties are used for a static definition of the element labelling. Each can contain a formatting sequence [▶ 617], e.g. %s. In online mode the sequence is replaced by the contents of the variables defined in "Text variables".

Text	Enter a text. It is used to label the element. The text can contain a formatting sequence, e.g. %s. The corresponding variable is defined in "Text variables". Note: The text can also be entered directly. If the element is selected in the visualization editor, an input field can be opened by pressing the space bar.
Tooltip	Enter a text. It is used as a tooltip for the element and appears in the visualization only in online mode when the cursor is placed over an element. The text can contain a formatting sequence, e.g. %s. The corresponding variable is defined in "Text variables".



Text properties

These properties are used for a static definition of the font. A dynamic definition of the font is possible in the category "Font variables".

Horizontal alignment	<p>Defines the horizontal alignment of the text through selection of:</p> <ul style="list-style-type: none"> • Left • Centered • Right
Vertical alignment	<p>Defines the vertical alignment of the text through selection of:</p> <ul style="list-style-type: none"> • Top • Centered • Bottom
Text format	<p>Defines the display of a text that is too long to be displayed completely in an element:</p> <ul style="list-style-type: none"> • Default – the text extends beyond the element. • Line break – the text is automatically wrapped. • Ellipsis - the text is displayed as far as possible and then truncated with "...".
Font:	<p>Defines the font through selection from predefined fonts:</p> <ul style="list-style-type: none"> • Standard • Heading • Large heading • Title • Note <p>Press  to opens the dialog for user-defined font properties.</p>
Font color	<p>Defines the font color for the element. Either from the selection list or via the dialog that opens when  is clicked.</p>

Absolute movement

The element can be moved by changing the x- and y-position (pixels) of the top left corner of the element via an integer variable. Absolute coordinate values are used here.

<p>Motion</p> <ul style="list-style-type: none"> • X • Y 	<p>X: The integer variable entered here defines the current x-position of the top left corner of the element (in pixels). It can be used to move the element in x-direction. (A positive value moves the element from left to right).</p> <p>Y: The integer variable entered here defines the current y-position of the top left corner of the element (in pixels). It can be used to move the element in y-direction. (A positive value moves the element from top to bottom).</p>
<p>Rotation</p>	<p>The integer variable entered here defines the angle (angular degrees) for rotating the element around a rotation point.</p> <p>Positive values = clockwise</p> <p>Note: In contrast to the behavior with 'internal rotation' (see below), the element itself does not rotate. Click on the element to show the rotation point . It can be moved by pressing and holding the mouse button.</p>
<p>Scaling</p>	<p>The integer variable entered here defines the current scaling factor (percent). The element size is adjusted linearly according to this value. The value is implicitly divides by 1000, so that it is not necessary to use REAL variables in order to shrink the element. The scaling always refers to the rotation point (center). Click on the element to show the rotation point . It can be moved by pressing and holding the mouse button.</p>

Relative movement

The element can be moved relative to its fixed position. The top left and bottom right edges of the element are moved in x- or y-direction by a value (pixels) defined by an integer variable. In contrast to an absolute movement, a relative position is defined, i.e. the distance to the original position. This function can be used to change the shape of the element. Positive values move the horizontal edges downwards and/or the vertical edges to the right.

Top left movement • X • Y	<ul style="list-style-type: none"> • X: Integer variable, the value of which indicates the number of pixels by which the top left corner is moved in x-direction. • Y: Integer variable, the value of which indicates the number of pixels by which the top left corner is moved in y-direction.
Bottom right movement • X • Y	<ul style="list-style-type: none"> • X: Integer variable, the value of which indicates the number of pixels by which the bottom right corner is moved in x-direction. • Y: Integer variable, the value of which indicates the number of pixels by which the bottom right corner is moved in y-direction.

Text variables

You can display text that is stored in a variable. To do this, first add a formatting sequence in the text that is defined under the "Texts" property. Then assign a variable. In online mode the [formatting sequence \[▶ 617\]](#) is replaced by the contents of the variables.

Example %f:

Enter "Result: %2.5f" in the "Texts" property. Enter "fValue" in the "Text variables" property. The variable must be a defined IEC variable. The element will then be labeled in online mode with "Result: 12.12345" if fValue = 12.1234567.

Text variable	Variable (of default data type) containing the information to be displayed. The type must match the formatting sequence in the "Texts" setting.
Tooltiptext	Variable of type String containing the tooltip text to be displayed. The entry in the property "Texts" must contain a formatting sequence.

Dynamic texts

These parameters are used to define dynamic texts originating from [text lists \[▶ 138\]](#). This enables [language change \[▶ 616\]](#), for example.

A further possibility to define a text dynamically is to supply the text via a string variable. (see the category "Text variables")

Text list	Name of the text list, as used in the project tree, as a string Example: 'TL_ErrorList'
Textindex	Index (ID) of the text, as defined in the text list, as a string. It can be specified directly in a static manner or as a string variable.
Tooltiptext	Index (ID) of the tooltip text, as defined in the text list, as a string. It can be specified directly in a static manner or as a string variable.

Font variables

These variables are used for dynamic font definitions for element texts via project variables. Static definitions are configured under "Text properties".

Font name	Specification of a variable of the type String that contains the font name that is to be used to label the element. (name specified as in the standard font dialog) Example: MAIN.sFont (sFont := 'Arial');
Size	Variable of type INT containing the size of the element text in pixels, as in the default dialog 'Font'. Example: MAIN.nHeight (nHeight := 16;)
Flags	Variable of type DWORD for defining the font display via one of the flag values listed below. A combined definition can be achieved by adding the respective flag values and specifying the sum. <ul style="list-style-type: none"> • 1: italics • 2: bold • 4: underlined • 8: deleted Example: MAIN.nFlag (nFlag := 6;) The text is shown bold and underlined.
Character set	The character set to be used for the font can be defined via the default font number. This number can be specified via a DWORD variable (see also the definition in the default font dialog)
Color	Variable of type DWORD for defining the color of the element text.
Flags for the text alignment	Variable of type DWORD for defining the text alignment via one of the flag values listed below. A combined definition can be achieved by adding the respective flag values and specifying the sum. <ul style="list-style-type: none"> • 0: top left • 1: centered horizontally • 2: right • 4: centered vertically • 8: below Example MAIN.nFlag (nFlag := 5;) The text is displayed centered horizontally and vertically.

Color variables

The color variables used for dynamic definition of the element colors via project variables of type DWORD. A color is defined based on a hexadecimal number consisting of red, green and blue (RGB) components. In addition, the variables are used to specify the transparency of the color (FF: fully opaque - 00: fully transparent). The DWORD has the following structure: 16#TTRRGGBB



The structure of the hexadecimal number differs from TwinCAT 2. In TwinCAT 3 it is possible to define the transparency of the color with the hexadecimal number in addition to the RGB proportions. The transparency is indicated by the first two digits after "16#". Colors with definitions starting with "16#00" are invisible, since they are fully transparent.

Example:

nFillColor := 16#FF8FE03F;

- FF: transparency (fully opaque)
- 8F: red
- E0: green
- 3F: blue

Color change	Boolean variable, which controls the switching of the element color between "normal state" (variable = FALSE) and "alarm state" (variable = TRUE).
Normal state	Variable of type DWORD for defining the element color. It overwrites the value that is currently defined in "Colors". The value in the project variable is used, if the variable defined in "Color change" is FALSE.
Alarm state	Variable of type DWORD for defining the element color in alarm state. It overwrites the value that is currently defined in "Colors". The value in the project variable is used, if the variable defined in "Color change" is TRUE.



Transparency is not supported under Windows CE.

Appearance variables

To be used for a dynamic definition of the contour appearance and the element filling. Static definitions are specified under "Appearance".

Line width	Variable of type integer for defining the line width of the element in pixels. It overwrites the fixed value that was specified under "Appearance".
Fill type	Variable of type DWORD for defining the element filling. The color specified under color variables can be displayed or ignored: <ul style="list-style-type: none"> • Variable value = 0: filled • Variable value > 0: invisible, i.e. no filling is visible
Line type	Variable of type DWORD for defining the contour. The following values correspond to the following line type: <ul style="list-style-type: none"> • 0: solid • 1: dashed • 2: dotted • 3: dotdash • 4: dash dot dot • 8: invisible. That is, the element is shown without a contour.

Toggle variable

This property can be used to switch between visualizations of a frame.

Variable	Integer variable whose value contains the ID of the visualization to be displayed. The ID of a visualization is determined by the order in the list of assigned visualizations in the Frame selection [▶ 617]. For example, the first entry in this list has the ID 0, the second entry the ID 1. The visualizations, which are assigned to a frame, can be toggled with a variable. The value (ID) of the visualization is determined by the order of this element in the list of the assigned visualizations in the dialog "Configuration of the frame visualizations". For integer values, the first entry in this list results in the value 0, the second entry 1 etc.
----------	---

State variables

These are dynamic definitions of the availability of the element in online mode.

Invisibility	Specification of a boolean variable. If this returns TRUE, the element is invisible in online mode.
Input disabled	Specification of a boolean variable. If TRUE is returned, inputs for the element have no effect. Also, the element itself is greyed out in the visualization, to indicate that no user inputs are possible. If the visualization uses the user management [▶ 393], the elements for user groups with access right "only visible" are greyed out.

Input configuration

Here you can define the consequential action that should be performed when the user makes an input in the element in online mode. As long as no follow-on actions are defined, "Configure..." appears in the Properties field. Click on "Configure..." to open the [Input Configuration \[► 406\]](#), where you can assign follow-on actions. Each input action can be assigned any number of follow-on actions.

The following input events are available for an element:

- OnDialogClosed
- OnMouseClicked
- OnMouseDown
- OnMouseEnter
- OnMouseLeave
- OnMouseMove
- OnMouseUp

OnDialogClosed	This event is triggered if one of the dialog boxes that were opened for user inputs is closed within a visualization. Note: This property is not limited to the element, for which it is configured, but applies within the entire visualization. It therefore does not respond to each dialog closing action. Currently there is no way to define such a property for the entire visualization. It therefore has to be assigned to one of its elements.
OnMouseClicked	This mouse event is triggered when the cursor points to an element and a full mouse click (pressing and releasing the mouse button) is executed on this item.
OnMouseDown	This mouse event is triggered when the mouse button is pressed while the cursor points to an element. It is irrelevant where on the visualization the mouse button is released again.
OnMouseEnter	This mouse event is triggered when the cursor is dragged to the element. It is irrelevant whether the mouse button is pressed or released.
OnMouseLeave	This mouse event is triggered when the cursor leaves the element. It is irrelevant whether the mouse button is pressed or released.
OnMouseMove	This mouse event is triggered when the cursor is moved within the element. It is irrelevant whether the mouse button is pressed or released.
OnMouseUp	This mouse event occurs when the mouse button is released on the element. The mouse button was pressed prior to that outside of the element.

Input configuration - hotkeys

A hotkey can be used to define a key or key combination and link it with a follow-on action (e.g. MouseDown, MouseUp), which is to be executed when a key event occurs (KeyDown, KeyUp). By default the MouseDown action is executed on KeyDown (press key) and the MouseUp action on KeyUp (release key). This can be useful if a visualization is to be operated both via mouse actions and keyboard entries, since the input actions then only have to be configured once. The key configuration for an element is also managed in the [hotkeys configuration \[► 392\]](#) for the visualization. Any changes are always synchronized between this and the element properties editor.

Key	Assigning a key. A selection list contains all currently supported keys, e.g. M.
Event(s)	Definition of the event to be executed, if the key or key and modifier are used. Possible values that are available in a selection list: <ul style="list-style-type: none"> • No action • MouseDown action when the key is pressed • MouseUp action when the key is released • MouseDown/MouseUp action when the key is pressed/released
Shift	If this option is enabled, the key has to be used in combination with the Shift key.
Control	If this option is enabled, the key has to be used in combination with the Ctrl key.
Alt	If this option is enabled, the key has to be used in combination with the Alt key.

Input configuration – Keys

"Keys" can be used to specify that the value of a boolean project variable is set depending on the mouse behavior for the event "Keys".

Variable	Boolean variable whose value is TRUE if the mouse button is pressed while the cursor points to the element. The value becomes FALSE again when the mouse button is released or the cursor leaves the element.
FALSE keys	If this option is enabled, the key behavior described above is reversed for the corresponding variable. That is, when the mouse button pressed, the variable value is set to FALSE. When the button is, the variable value is set to TRUE.
Switch on enter if mouse trapped	If this option is enabled, the variable value behaves as described under 'Variables', as long as the cursor is within the element area. If the cursor leaves the element area while the mouse button is pressed, the variable is set to FALSE. However, it is automatically set to TRUE again, if the cursor returns to the element area without the mouse button being released. In other words, the system takes into account that the mouse is "trapped" as long as the mouse button is pressed, even if the cursor leaves the element area.

Input configuration - Switching

"Switching" can be used to specify that for this event the value of a boolean project variable is set depending on the mouse behavior.

Variable	Boolean variable whose value switches between TRUE and FALSE with each mouse click on the element. No switching occurs if the cursor is moved away from the element while the mouse button is pressed. Can be used to cancel a switching action.
Switch on release, if mouse trapped	If this option is enabled, the variable value behaves as described under 'Variables', as long as the cursor is within the element area. If the cursor leaves the element area while the mouse button is pressed and the mouse button is then released, the variable is switched anyway. In other words, the system takes into account that the mouse is "trapped" as long as the mouse button is pressed, even if the cursor leaves the element area.

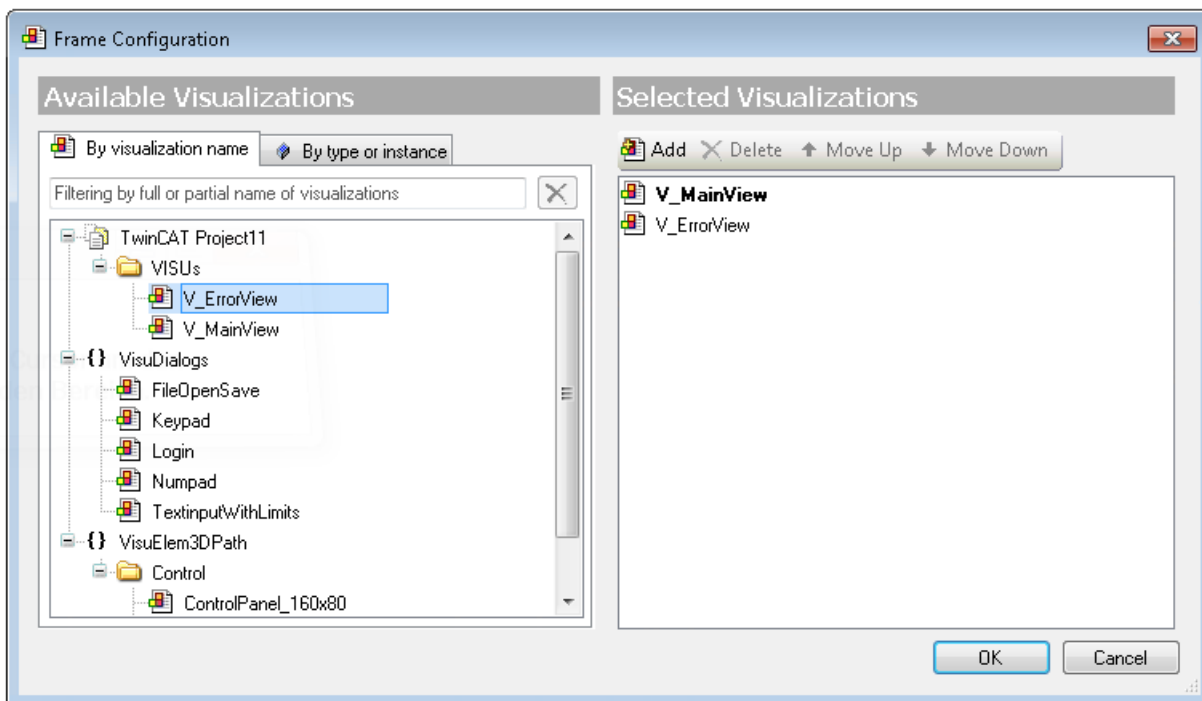
Access rights

This setting relates to the access rights for the individual element. Click to open the [Access rights dialog](#) [▶ 420]. The setting is only available if a [user management](#) [▶ 393] was added to the PLC project. The following status messages are available:

Not set. All rights.	The default message is set, if the element is shown as available for all groups.
Rights are issued: Limited rights.	The message is set, if the element is shown with limited behavior for at least one group.

15.8.3.6.1 Frame selection

This dialog is used to configure a [frame](#) [▶ 516] element. It can be used to select one or several [visualization pages](#) [▶ 402] to be referenced via the frame element.

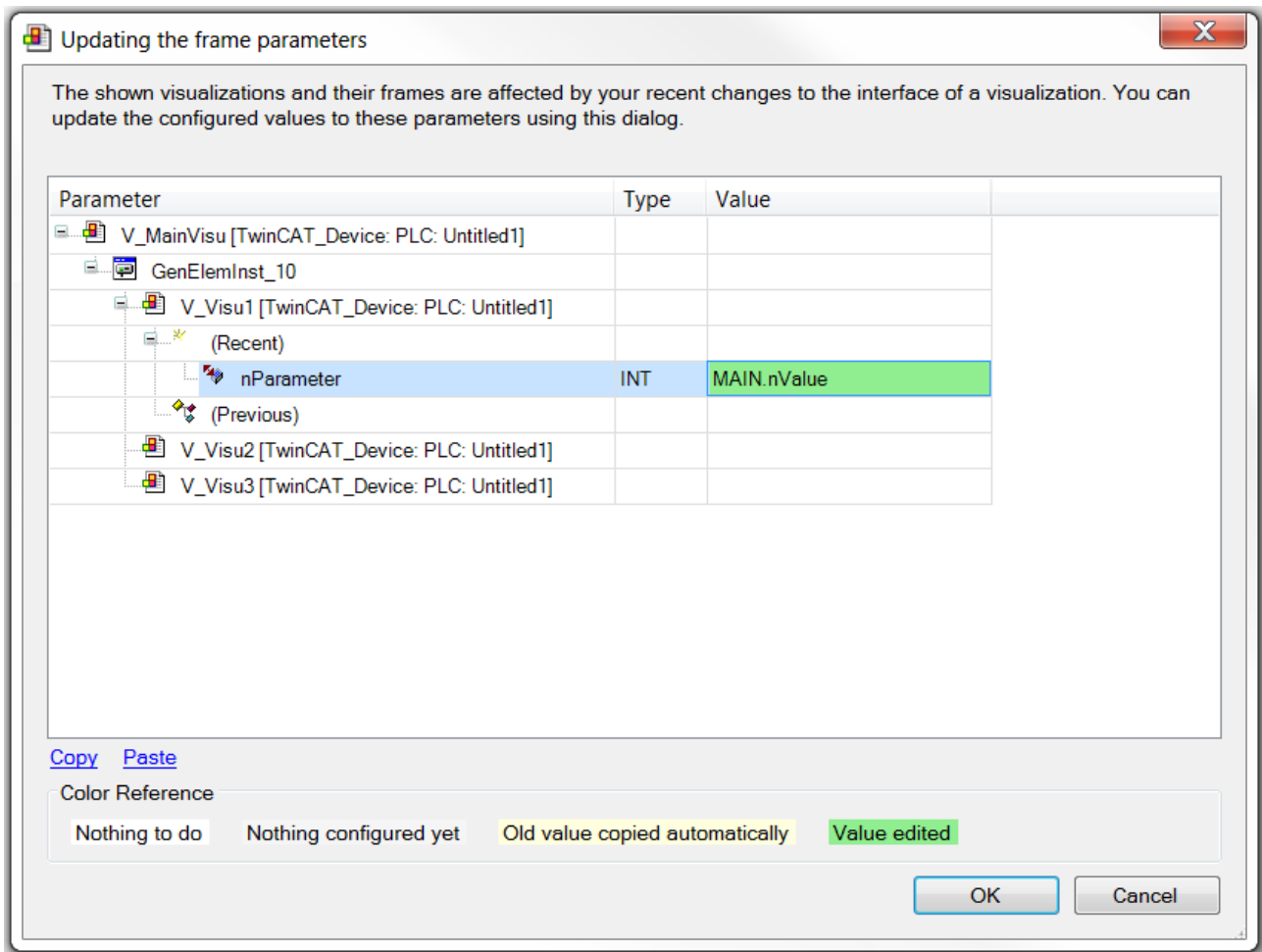


The visualization pages or objects available in the project are shown on the left in the dialog. Select the ones to be referenced in the frame. Double-click or use the "Add" button to add them to the list of selected visualization pages on the right-hand side of the dialog. Selected visualizations can be removed from the list by double-clicking or via the "Delete" button. Multiple selection of visualization pages is possible for adding or deleting. The order within the list can be changed with the "Move Up" and "Move Down" buttons.

The order of the selected visualization objects from top to bottom is determined by automatically generated implicit visualization index numbers. The object at the top is assigned "0", the following objects "1", "2", etc. The index numbers are required for the configuration of the [switching function](#) [▶ 414] for another element. Initially the visualization page with the index "0" is shown.

15.8.3.6.2 Updating the frame parameters

The dialog appears if the parameters of a referenced [visualization page](#) [▶ 402] are changed in the [interface editor](#) [▶ 378]. The editor can be used to add or update [frame parameters](#) [▶ 380].



How to use the dialog is described under [interface editor \[▶ 378\]](#).

15.8.4 Lamps/Switches/Bitmaps

15.8.4.1 Image switcher

The element image switcher displays an image consisting of [three referenced images \[▶ 528\]](#). If a mouse action occurs while the visualization is running, the image that is displayed changes. The consequences of mouse actions are configurable via the element behavior. Further information on the options for the integration of images in the visualization can be found in the section "[Images \[▶ 620\]](#)" in the "Application hints".

Properties editor


The properties of a visualization element - except [alignment and order \[▶ 377\]](#) - can all be configured in the [properties editor \[▶ 385\]](#). By default, this editor opens next to the visualization editor, or it can be opened explicitly via the "Properties" command (which can be found in the View menu as standard).

A property can be modified by editing the field "Value". To this end, an input field, a selection list, a dialog or checkbox that can be activated is provided in this field, depending on the element type. The value field opens

- after a double-click,
- after a single click in a selected field,
- via the space bar, if the field was already selected.

If a variable is assigned,

- simply enter its name.

- Use the  button to open the input assistant for selecting a variable. The Variables category lists all variables that have already been defined in the project.

Working in the list of properties can be made easier with the aid of default, sorting and filter functions.

Element properties

All element properties and their descriptions are listed below.

Element name	The element name can be changed. Standard name is "GenElemInst_x". "x" stands for a sequential number.
Element type	The element type is entered here. For three element groups it is possible to switch between the corresponding elements by changing the element type: <ul style="list-style-type: none"> • rectangle, rounded rectangle and ellipse [► 475] • Polygon, polyline or Bézier curve [► 490] • dip switch, power switch, push switch, push switch with LED, rocker switch [► 532]

Position

Here you can define the position (X/Y coordinates) and size (width and height) of the element in pixels. The origin is in the top left corner of the window. The positive x-axis is on the right, the positive y-axis runs downwards. If the values edited, the displayed element is simultaneously modified in the [visualization editor \[► 376\]](#).

X	Horizontal position in pixels – X=0 is the left edge of the window.
Y	Vertical position in pixels – Y=0 is the upper edge of the window.
Width	Width of the element in pixels
Height	Height of the element in pixels

Auxiliary setting

Variable	Assignment of a boolean variable, whose state changes according to the user input. Depending on the element behavior the variable is TRUE, as long as the mouse button is pressed (image button), or the value changes with each mouse click (image switcher).
-----------------	--

Image settings

To keep the entries for the image names uniform, the name of the image pool should be specified as a prefix. This is not necessary if the images are managed via the GlobalImagePool, since this is in any case searched first.

Image on	ID of a bitmap from the image pool. The image is shown in On state.
Image off	ID of a bitmap from the image pool. The image is shown in Off state.
Image pressed	ID of a bitmap from the image pool. At runtime the visualization shows the referenced image, as soon as the element receives the mouse action "right mouse button pressed". Requirement: Element behavior is image switcher.
Transparency	If this option is enabled, the image is drawn fully transparent for a section defined by the transparency color.
Transparency color	Here you can set the color that is shown fully transparent in the image. Example: If the image background is white and the transparency color is also white, the background is drawn fully transparent. Requirement: Transparency is enabled.
Scaling type	Here you can specify how the element responds to changes in the frame size: <ul style="list-style-type: none"> • Isotropic: Height and width of the image are changed with the frame. Their original proportions are retained, however. • Anisotropic: The image fills the whole frame, irrespective of its proportions. • Unscaled: The image size is specified and remains unchanged, if the frame size changes.
Horizontal alignment	Only available if the "Expert" option is activated for the properties editor and if the image scaling type is "isotropic". This is used to explicitly maintain the horizontal alignment to another element (as defined in the basic visualization), if the visualization is used within a scaled frame. See above: "Isotropic" setting <ul style="list-style-type: none"> • Left • Centered • Right
Vertical alignment	Only available if the "Expert" option is activated for the properties editor and if the image scaling type is "isotropic". This is used to explicitly maintain the vertical alignment to another element (as defined in the basic visualization), if the visualization is used within a scaled frame. See above: "Isotropic" setting <ul style="list-style-type: none"> • Top • Centered • Bottom
Element behavior	Here you can select the element behavior: <ul style="list-style-type: none"> • Image switcher: The state of the element and therefore the image and the variable toggle with each mouse click. • Image button: 'Image on' is displayed and the variable is TRUE, as long as the mouse button is pressed. In this state the image that is referenced in "Image pressed" is not used.
FALSE keys	If this option is enabled, the variable is set to FALSE by pressing the mouse button. Otherwise the variable is set to TRUE. Requirement: Image switcher is set in element behavior.

Texts

Tooltip	Here you can set the text to be used as tooltip for the element. It only appears in the visualization at runtime.
---------	---

State variables

These are dynamic definitions of the availability of the element in online mode.

Invisibility	Specification of a boolean variable. If this returns TRUE, the element is invisible in online mode.
Input disabled	Specification of a boolean variable. If TRUE is returned, inputs for the element have no effect. Also, the element itself is greyed out in the visualization, to indicate that no user inputs are possible. If the visualization uses the user management [► 393] , the elements for user groups with access right "only visible" are grayed out.

Access rights

This setting relates to the access rights for the individual element. Click to open the [Access rights dialog \[► 420\]](#). The setting is only available if a [user management \[► 393\]](#) was added to the PLC project. The following status messages are available:

Not set. All rights.	The default message is set, if the element is shown as available for all groups.
Rights are issued: Limited rights.	The message is set, if the element is shown with limited behavior for at least one group.

15.8.4.2 Lamp

The lamp comes on if the assigned variable is set. The lamp color can be changed in the lamp [settings \[► 532\]](#).




Properties editor

The properties of a visualization element - except [alignment and order \[► 377\]](#) - can all be configured in the [properties editor \[► 385\]](#). By default, this editor opens next to the visualization editor, or it can be opened explicitly via the "Properties" command (which can be found in the View menu as standard).

A property can be modified by editing the field "Value". To this end, an input field, a selection list, a dialog or checkbox that can be activated is provided in this field, depending on the element type. The value field opens

- after a double-click,
- after a single click in a selected field,
- via the space bar, if the field was already selected.

If a variable is assigned,

- simply enter its name.
- Use the  button to open the input assistant for selecting a variable. The Variables category lists all variables that have already been defined in the project.

Working in the list of properties can be made easier with the aid of default, sorting and filter functions.

Element properties

All element properties and their descriptions are listed below.

Element name	The element name can be changed. Standard name is "GenElemInst_x". "x" stands for a sequential number.
Element type	The element type is entered here. For three element groups it is possible to switch between the corresponding elements by changing the element type: <ul style="list-style-type: none"> • rectangle, rounded rectangle and ellipse [► 475] • Polygon, polyline or Bézier curve [► 490] • dip switch, power switch, push switch, push switch with LED, rocker switch [► 532]

Position

Here you can define the position (X/Y coordinates) and size (width and height) of the element in pixels. The origin is in the top left corner of the window. The positive x-axis is on the right, the positive y-axis runs downwards. If the values edited, the displayed element is simultaneously modified in the [visualization editor \[► 376\]](#).

X	Horizontal position in pixels – X=0 is the left edge of the window.
Y	Vertical position in pixels – Y=0 is the upper edge of the window.
Width	Width of the element in pixels
Height	Height of the element in pixels

Auxiliary setting

Variable	Boolean variable whose value switches the state of the lamp. With TRUE the lamp is switched on and depicted shining.
-----------------	--

Image setting

Scaling type	Here you can specify how the element responds to changes in the frame size: <ul style="list-style-type: none"> • Isotropic: Height and width of the image are changed with the frame. Their original proportions are retained, however. • Anisotropic: The image fills the whole frame, irrespective of its proportions. • Unscaled: The image size is specified and remains unchanged, if the frame size changes.
Horizontal alignment	Only available if the "Expert" option is activated for the properties editor and if the image scaling type is "isotropic". This is used to explicitly maintain the horizontal alignment to another element (as defined in the basic visualization), if the visualization is used within a scaled frame. See above: "Isotropic" setting <ul style="list-style-type: none"> • Left • Centered • Right
Vertical alignment	Only available if the "Expert" option is activated for the properties editor and if the image scaling type is "isotropic". This is used to explicitly maintain the vertical alignment to another element (as defined in the basic visualization), if the visualization is used within a scaled frame. See above: "Isotropic" setting <ul style="list-style-type: none"> • Top • Centered • Bottom

Texts

Tooltip	Here you can set the text to be used as tooltip for the element. It only appears in the visualization at runtime.
---------	---

Background

Image	Here you can select the lamp color: <ul style="list-style-type: none"> • Yellow • Red • Green • Blue • Grey
-------	--

State variables

These are dynamic definitions of the availability of the element in online mode.

Invisibility	Specification of a boolean variable. If this returns TRUE, the element is invisible in online mode.
Input disabled	Specification of a boolean variable. If TRUE is returned, inputs for the element have no effect. Also, the element itself is greyed out in the visualization, to indicate that no user inputs are possible. If the visualization uses the <u>user management</u> [▶ 393], the elements for user groups with access right "only visible" are greyed out.

Access rights

This setting relates to the access rights for the individual element. Click to open the Access rights dialog [[▶ 420](#)]. The setting is only available if a user management [[▶ 393](#)] was added to the PLC project. The following status messages are available:

Not set. All rights.	The default message is set, if the element is shown as available for all groups.
Rights are issued: Limited rights.	The message is set, if the element is shown with limited behavior for at least one group.

15.8.4.3 Dip switch, power switch, push switch, push switch LED, rocker switch

The switch is used to set the value of a boolean variable.



Properties editor


The properties of a visualization element - except alignment and order [[▶ 377](#)] - can all be configured in the properties editor [[▶ 385](#)]. By default, this editor opens next to the visualization editor, or it can be opened explicitly via the "Properties" command (which can be found in the View menu as standard).

A property can be modified by editing the field "Value". To this end, an input field, a selection list, a dialog or checkbox that can be activated is provided in this field, depending on the element type. The value field opens

- after a double-click,
- after a single click in a selected field,
- via the space bar, if the field was already selected.

If a variable is assigned,

- simply enter its name.

- Use the  button to open the input assistant for selecting a variable. The Variables category lists all variables that have already been defined in the project.

Working in the list of properties can be made easier with the aid of default, sorting and filter functions.

Element properties

All element properties and their descriptions are listed below.

Element name	The element name can be changed. Standard name is "GenElemInst_x". "x" stands for a sequential number.
Element type	The element type is entered here. For three element groups it is possible to switch between the corresponding elements by changing the element type: <ul style="list-style-type: none"> • rectangle, rounded rectangle and ellipse [► 475] • Polygon, polyline or Bézier curve [► 490] • dip switch, power switch, push switch, push switch with LED, rocker switch [► 532]

Position

Here you can define the position (X/Y coordinates) and size (width and height) of the element in pixels. The origin is in the top left corner of the window. The positive x-axis is on the right, the positive y-axis runs downwards. If the values edited, the displayed element is simultaneously modified in the [visualization editor \[► 376\]](#).

X	Horizontal position in pixels – X=0 is the left edge of the window.
Y	Vertical position in pixels – Y=0 is the upper edge of the window.
Width	Width of the element in pixels
Height	Height of the element in pixels

Auxiliary setting

Variable	Boolean variable whose value changes according to the user entry. Depending on the element behavior, the variable is TRUE as long as the mouse button is pressed (momentary) or the value changes with each mouse click (changeover).
-----------------	---

Image setting

Scaling type	Here you can specify how the element responds to changes in the frame size: <ul style="list-style-type: none"> • Isotropic: Height and width of the image are changed with the frame. Their original proportions are retained, however. • Anisotropic: The image fills the whole frame, irrespective of its proportions. • Unscaled: The image size is specified and remains unchanged, if the frame size changes.
Horizontal alignment	Only available if the "Expert" option is activated for the properties editor and if the image scaling type is "isotropic". This is used to explicitly maintain the horizontal alignment to another element (as defined in the basic visualization), if the visualization is used within a scaled frame. See above: "Isotropic" setting <ul style="list-style-type: none"> • Left • Centered • Right
Vertical alignment	Only available if the "Expert" option is activated for the properties editor and if the image scaling type is "isotropic". This is used to explicitly maintain the vertical alignment to another element (as defined in the basic visualization), if the visualization is used within a scaled frame. See above: "Isotropic" setting <ul style="list-style-type: none"> • Top • Centered • Bottom

State variables

These are dynamic definitions of the availability of the element in online mode.

Invisibility	Specification of a boolean variable. If this returns TRUE, the element is invisible in online mode.
Input disabled	Specification of a boolean variable. If TRUE is returned, inputs for the element have no effect. Also, the element itself is greyed out in the visualization, to indicate that no user inputs are possible. If the visualization uses the user management [▶ 393], the elements for user groups with access right "only visible" are greyed out.

Texts

Tooltip	Here you can set the text to be used as tooltip for the element. It only appears in the visualization at runtime.
---------	---

Background

Image	Here you can select the lamp color: <ul style="list-style-type: none"> • Yellow • Red • Green • Blue • Grey
-------	--

Access rights

This setting relates to the access rights for the individual element. Click to open the [Access rights dialog](#) [[▶ 420](#)]. The setting is only available if a [user management](#) [[▶ 393](#)] was added to the PLC project. The following status messages are available:

Not set. All rights.	The default message is set, if the element is shown as available for all groups.
Rights are issued: Limited rights.	The message is set, if the element is shown with limited behavior for at least one group.

15.8.4.4 Rotary switch

The rotary switch is used to set the value of a boolean variable.




Properties editor

The properties of a visualization element - except alignment and order [[▶ 377](#)] - can all be configured in the properties editor [[▶ 385](#)]. By default, this editor opens next to the visualization editor, or it can be opened explicitly via the "Properties" command (which can be found in the View menu as standard).

A property can be modified by editing the field "Value". To this end, an input field, a selection list, a dialog or checkbox that can be activated is provided in this field, depending on the element type. The value field opens

- after a double-click,
- after a single click in a selected field,
- via the space bar, if the field was already selected.

If a variable is assigned,

- simply enter its name.
- Use the  button to open the input assistant for selecting a variable. The Variables category lists all variables that have already been defined in the project.

Working in the list of properties can be made easier with the aid of default, sorting and filter functions.

Element properties

All element properties and their descriptions are listed below.

Element name	The element name can be changed. Standard name is "GenElemInst_x". "x" stands for a sequential number.
Element type	The element type is entered here. For three element groups it is possible to switch between the corresponding elements by changing the element type: <ul style="list-style-type: none"> • <u>rectangle, rounded rectangle and ellipse</u> [▶ 475] • <u>Polygon, polyline or Bézier curve</u> [▶ 490] • <u>dip switch, power switch, push switch, push switch with LED, rocker switch</u> [▶ 532]

Position

Here you can define the position (X/Y coordinates) and size (width and height) of the element in pixels. The origin is in the top left corner of the window. The positive x-axis is on the right, the positive y-axis runs downwards. If the values edited, the displayed element is simultaneously modified in the visualization editor [[▶ 376](#)].

X	Horizontal position in pixels – X=0 is the left edge of the window.
Y	Vertical position in pixels – Y=0 is the upper edge of the window.
Width	Width of the element in pixels
Height	Height of the element in pixels

Auxiliary setting

Variable	Boolean variable whose value changes according to the user entry. Depending on the element behavior, the variable is TRUE as long as the mouse button is pressed (momentary) or the value changes with each mouse click (changeover).
-----------------	---

Image settings

Scaling type	Here you can specify how the element responds to changes in the frame size: <ul style="list-style-type: none"> • Isotropic: Height and width of the image are changed with the frame. Their original proportions are retained, however. • Anisotropic: The image fills the whole frame, irrespective of its proportions. • Unscaled: The image size is specified and remains unchanged, if the frame size changes.
Horizontal alignment	Only available if the "Expert" option is activated for the properties editor and if the image scaling type is "isotropic". This is used to explicitly maintain the horizontal alignment to another element (as defined in the basic visualization), if the visualization is used within a scaled frame. See above: "Isotropic" setting <ul style="list-style-type: none"> • Left • Centered • Right
Vertical alignment	Only available if the "Expert" option is activated for the properties editor and if the image scaling type is "isotropic". This is used to explicitly maintain the vertical alignment to another element (as defined in the basic visualization), if the visualization is used within a scaled frame. See above: "Isotropic" setting <ul style="list-style-type: none"> • Top • Centered • Bottom
Element behavior	Here you can select the element behavior: <ul style="list-style-type: none"> • Image switcher: The state of the element and therefore the image and the variable toggle with each mouse click. • Image button: 'Image on' is displayed and the variable is TRUE, as long as the mouse button is pressed. In this state the image that is referenced in "Image pressed" is not used.
FALSE keys	If this option is enabled, the variable is set to FALSE by pressing the mouse button. Otherwise the variable is set to TRUE. Requirement: Image switcher is set in element behavior.
Alignment	<ul style="list-style-type: none"> • Top: The switch points upwards. • Sideways: The switch points sideways.
Color change	If the option is not enabled, the switch is lit if it is switched on. If the option is not enabled, the switch is not lit, even if it is switched on.

Texts

Tooltip	Here you can set the text to be used as tooltip for the element. It only appears in the visualization at runtime.
---------	---

State variables

These are dynamic definitions of the availability of the element in online mode.

Invisibility	Specification of a boolean variable. If this returns TRUE, the element is invisible in online mode.
Input disabled	Specification of a boolean variable. If TRUE is returned, inputs for the element have no effect. Also, the element itself is greyed out in the visualization, to indicate that no user inputs are possible. If the visualization uses the user management [▶ 393] , the elements for user groups with access right "only visible" are grayed out.

Background

Image	Here you can select the lamp color: <ul style="list-style-type: none"> • Yellow • Red • Green • Blue • Grey
-------	--

Access rights

This setting relates to the access rights for the individual element. Click to open the [Access rights dialog \[▶ 420\]](#). The setting is only available if a [user management \[▶ 393\]](#) was added to the PLC project. The following status messages are available:

Not set. All rights.	The default message is set, if the element is shown as available for all groups.
Rights are issued: Limited rights.	The message is set, if the element is shown with limited behavior for at least one group.

15.8.5 Measurement controls

15.8.5.1 Bar display

This element can be used to add a bar display to the visualization. The bar display has a predefined design, for which the [background color \[▶ 538\]](#) can be changed. Optional, this design can be replaced by a user-specified background image. The display orientation can be [changed \[▶ 539\]](#) from horizontal to vertical, and the bar can be subdivided into [color areas \[▶ 540\]](#).



Properties editor


The properties of a visualization element - except [alignment and order \[▶ 377\]](#) - can all be configured in the [properties editor \[▶ 385\]](#). By default, this editor opens next to the visualization editor, or it can be opened explicitly via the "Properties" command (which can be found in the View menu as standard).

A property can be modified by editing the field "Value". To this end, an input field, a selection list, a dialog or checkbox that can be activated is provided in this field, depending on the element type. The value field opens

- after a double-click,
- after a single click in a selected field,
- via the space bar, if the field was already selected.

If a variable is assigned,

- simply enter its name.

- Use the  button to open the input assistant for selecting a variable. The Variables category lists all variables that have already been defined in the project.

Working in the list of properties can be made easier with the aid of default, sorting and filter functions.

Element properties

All element properties and their descriptions are listed below.

Element name	The element name can be changed. Standard name is "GenElemInst_x". "x" stands for a sequential number.
Element type	The element type is entered here. For three element groups it is possible to switch between the corresponding elements by changing the element type: <ul style="list-style-type: none"> • rectangle, rounded rectangle and ellipse [► 475] • Polygon, polyline or Bézier curve [► 490] • dip switch, power switch, push switch, push switch with LED, rocker switch [► 532]

Auxiliary setting

Variable	Numeric variable, whose value is displayed as a bar length.
----------	---

Position

Here you can define the position (X/Y coordinates) and size (width and height) of the element in pixels. The origin is in the top left corner of the window. The positive x-axis is on the right, the positive y-axis runs downwards. If the values edited, the displayed element is simultaneously modified in the [visualization editor \[► 376\]](#).


X	Horizontal position in pixels – X=0 is the left edge of the window.
Y	Vertical position in pixels – Y=0 is the upper edge of the window.
Width	Width of the element in pixels
Height	Height of the element in pixels

Background

If no user-specified background image is used, the following properties are available:

Background color	Select a color for the bar: <ul style="list-style-type: none"> • Yellow • Red • Green • Blue • Grey
------------------	--


If a user-specified background image is used, the following properties are available:

Image	Here you can assign an image from an image pool by specifying the name of the image file or its ID.
Transparency color	For images with transparent background a color can be selected that is to be shown transparent. The  button opens the color selection dialog.



Bars

Chart type	The drop-down list offers various options for positioning bars and scales: <ul style="list-style-type: none"> • Scale beside bar • Scale in bar • Bar in scale • No scale
Alignment	The bar can be aligned horizontally or vertically. The alignment is derived from the ratio between the width and height and cannot be edited here. It can be changed in the visualization editor [▶ 376] by "gripping" a corner point of the element with the mouse and dragging it horizontally or vertically.
Direction of movement	If the alignment is horizontal, there is a choice between: <ul style="list-style-type: none"> • Left to right • Right to left If the alignment is vertical, there is a choice between: <ul style="list-style-type: none"> • From bottom to top • From top to bottom

Scale

Scale start	Lower scale limit value
Scale end	Upper scale limit value
Main scale	Distance between two lines of the coarse scale
Sub scale	Distance between two lines on the fine scale. The value can be set to 0 if a further subdivision of the coarse scale is not desired.
Scale line width	Width of the scale line in pixels
Scale color	The color can be set via the selection list or via the  button from the standard color selection dialog.
Scale in 3D	If this option is enabled, the scale is shown 3-dimensionally.
Element frame	If this option is enabled, a frame is drawn around the bar display.

Labelling

Unit	The entered text is used as element label. It is shown below the center of the scale and can be used to specify the unit of the scale, for example.
Font	Here you can set the font for the unit and the scale: <ul style="list-style-type: none"> • Standard • Heading • Large • Title • Note Click the  button to open a dialog for user-defined font property settings.
Scale format (C-syntax)	Use the C syntax to specify the formatting of the scale label. For example, entering the string "%3.2f s" in this field results in scale labels with 3 digits, 2 of which are decimal places, followed by the letter "s".
Max. text width of labels	Value specifying the maximum width of the scale label. This value is generally set correctly automatically. Only use this setting option if the automatic adjustment does not lead to the required result.
Text height of labels	Value specifying the height of the scale label. This value is generally set correctly automatically. Only use this setting option if the automatic adjustment does not lead to the required result.
Font color	The color for the label can be set via the selection list or via the  button from the standard color selection dialog.

Positioning


Horizontal shift	Distance in pixels between the scale or the bar and the horizontal element frame
Vertical shift	Distance in pixels between the scale or the bar and the vertical element frame
Horizontal scaling	Factor for increasing (negative value) or reducing (positive value) the scale or the bar in horizontal direction
Vertical scaling	Factor for increasing (negative value) or reducing (positive value) the scale or the bar in vertical direction

Colors

A color is defined based on a hexadecimal number consisting of red / green / blue (RGB) components. For each of these three colors, 256 (0-255) values are available, which can be entered here statically. The color can be selected from a selection list or via the color selection dialog, which can be opened via the button



. In addition, the level of transparency can be set for each color (0: fully transparent, 255: fully opaque).

Graph color	Bar color
Bar background	If this option is enabled, the background color for the bar display is changed to black. Otherwise it is white.
Frame color	Color of the frame around the bar display, if the element frame checkbox is ticked
Use color areas	If this option is enabled, color areas are displayed as defined under Color areas.
Switch whole color	If this option is enabled, the whole bar switches its color, if the variable value falls below the start value or exceeds the end value of the scale.
Use color gradient for bar	If this option is enabled, the bar is shown with a color gradient.
Color area marking	Here you can select in which direction the color areas point. The following settings are available: <ul style="list-style-type: none"> • No marking • Forward marking • Backwards marking
Color areas <ul style="list-style-type: none"> • Areas <ul style="list-style-type: none"> ◦ [n] 	Click on the  Create new button to generate a new color area. For each color area an area is created that covers the corresponding settings. [n]: The number indexes the area. Clicking "Delete" deletes the corresponding color area and its settings.

Area [n]

Begin of area	Start of the color range. It must lie within the defined <u>scale</u> [▶ 539].
End of area	End of the color range. It must lie within the defined <u>scale</u> [▶ 539].
Color	Color of the bar area



Color gradient and transparency are not supported under Windows CE.

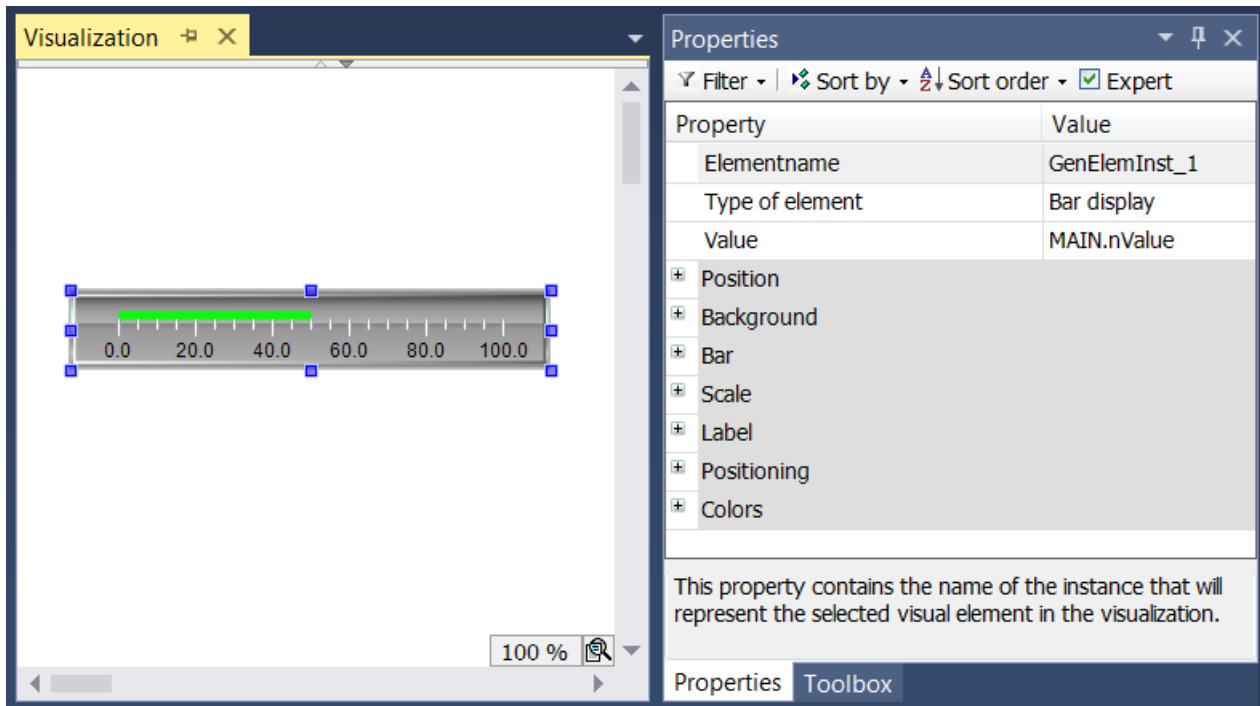
Access rights


This setting relates to the access rights for the individual element. Click to open the Access rights dialog [▶ 420]. The setting is only available if a user management [▶ 393] was added to the PLC project. The following status messages are available:

Not set. All rights.	The default message is set, if the element is shown as available for all groups.
Rights are issued: Limited rights.	The message is set, if the element is shown with limited behavior for at least one group.

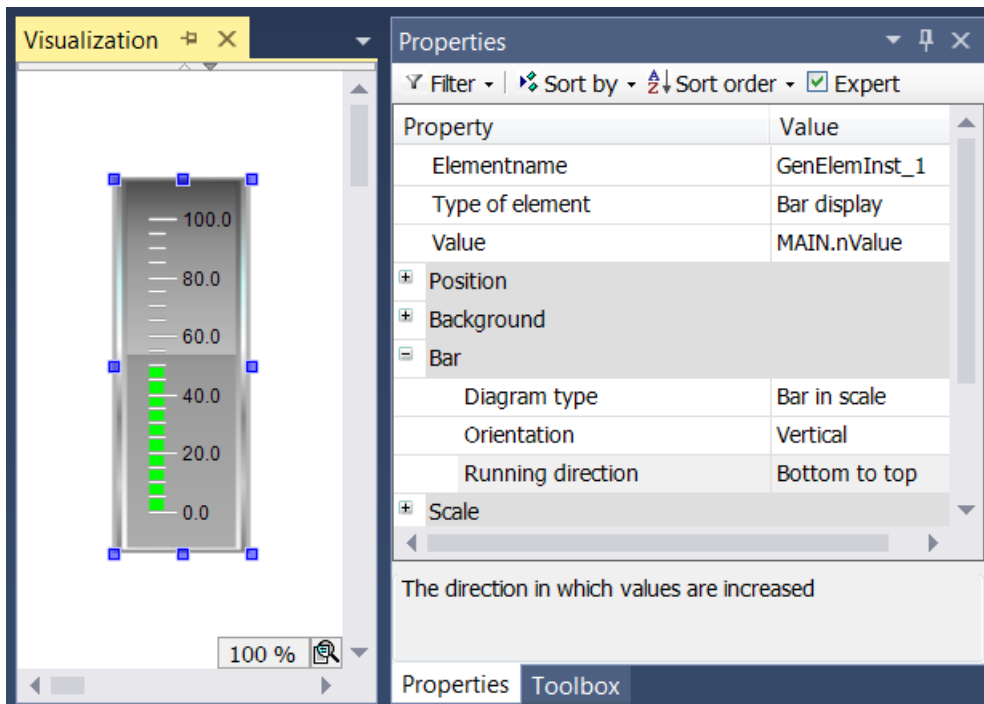
15.8.5.1.1 Configuration of a bar display

The following section explains the configuration of a pointer instrument, based on an example.



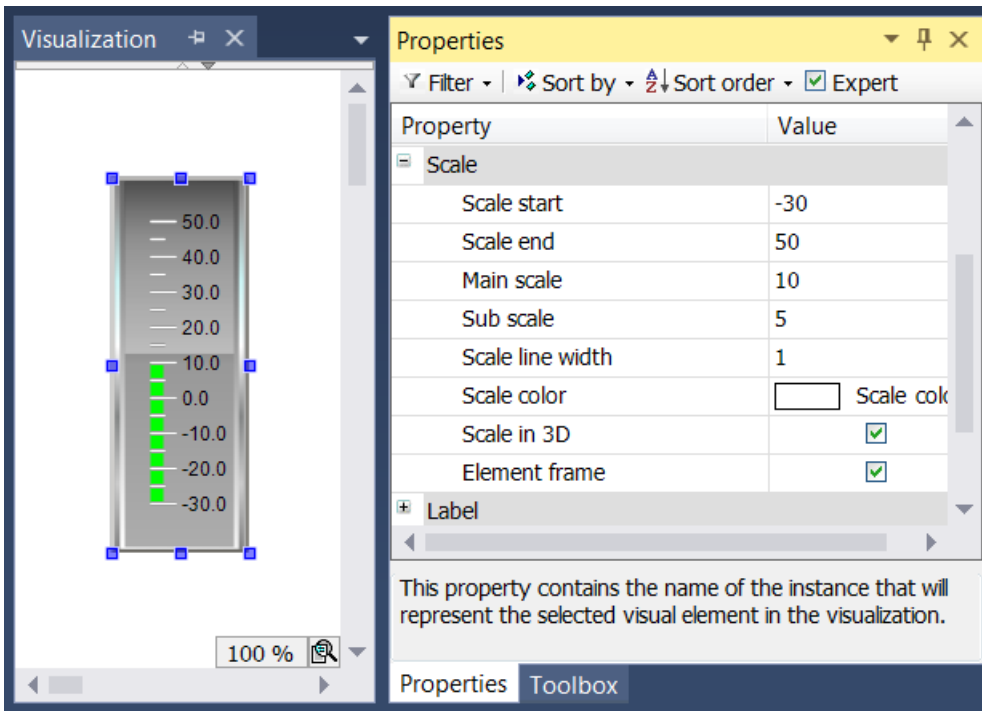
The input variable to be linked – in the example a variable with the name "nValue" – must be specified in the element properties of the 'Bar display'. After clicking in the input field for the property "Variable" the button  is available, which can be used to browse for the variable within the project.

The orientation and position of the bar relative to the scale can be set in the "Bar" section.

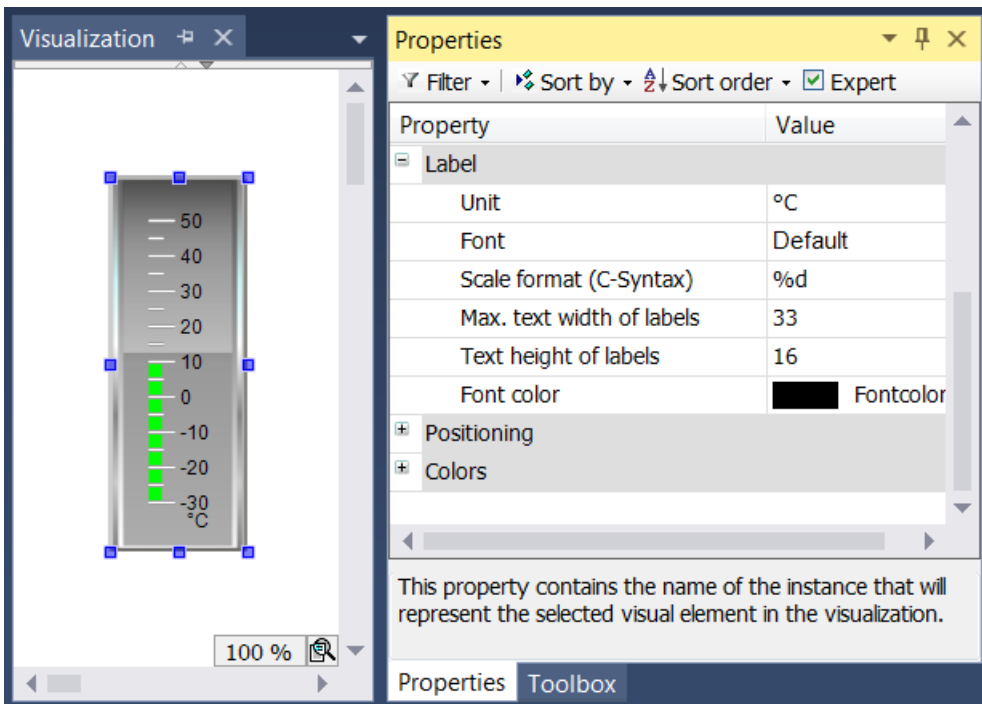


In the example the bar orientation was changed from "Horizontal" to "Vertical" by changing the aspect ratio from width to height. This setting also changes the possible information for the "Running direction". Instead of "left to right" or "right to left", you can now choose between "bottom to top" or "top to bottom". The position of the bar relative to the scale can be determined by setting the "Diagram type".

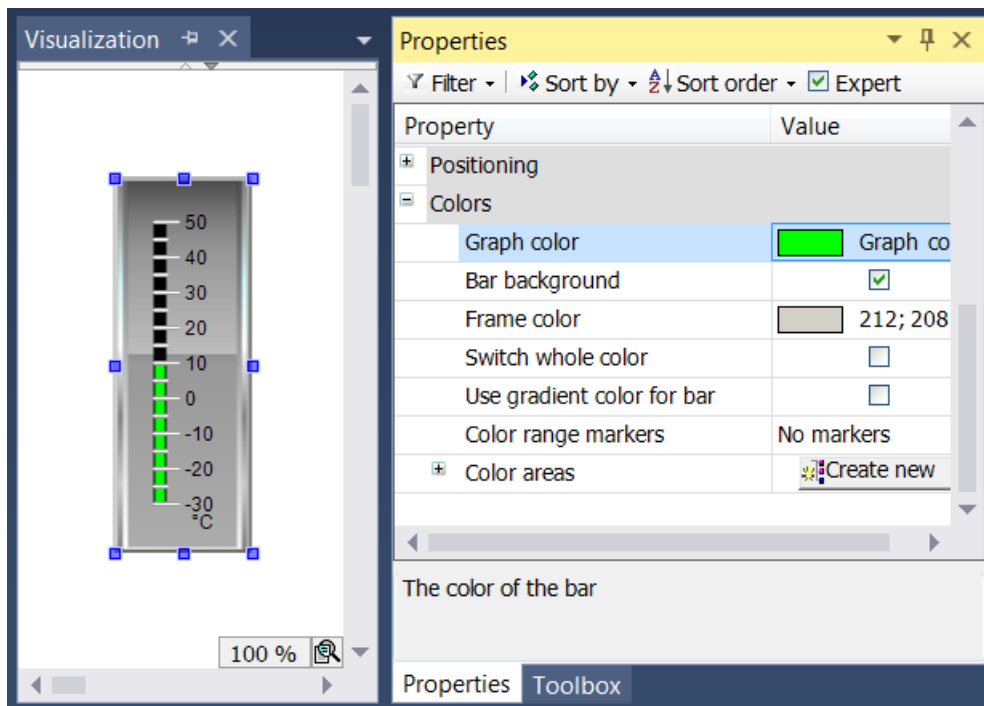
The "Scale" section is intended for setting the range of the scale and the main scaling and the subscaling.




The scale range is limited by the values specified in "Scale start" and "Scale end". The value of "Scale start" must be lower than that of "Scale end". The values for "Main scale" and "Sub scale" can be set to 0, in order to disable the display of the scale lines. If the value for the main scale is set to 0, no scale lines are drawn, irrespective of the value set for the sub scale. If the value for the sub scale is set to 0, only the scale lines for the main scale are drawn. The "Element frame" checkbox is ticked, because in the example we want a frame drawn around the element.




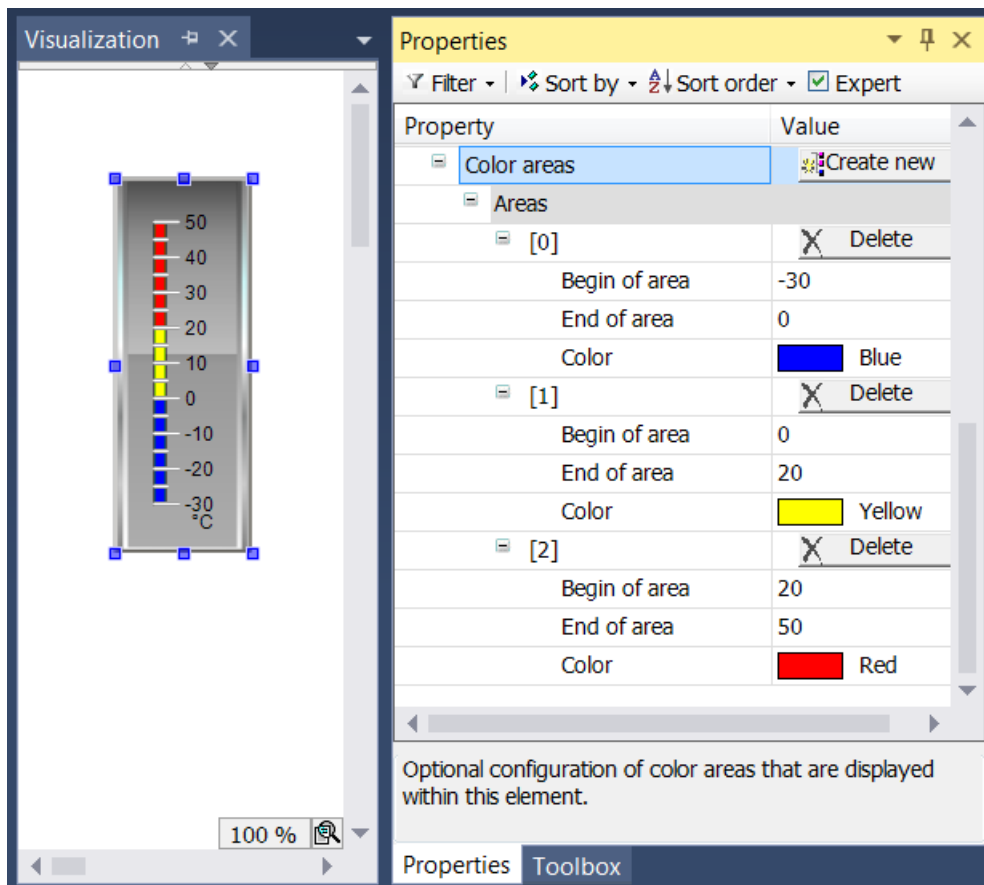
The "Label" section is used for specifying the bar label properties. The "Unit" entry is used to specify the unit. It is displayed centered below the bar. Once a suitable font and font color has been selected, the label format can be adjusted. Numerical values must be specified according to the C syntax. Use "%d" for integer values and "%.Xf" for floating-point numbers, whereby "X" should be replaced by the required number of decimal places.



Finally, the element color can be specified in the "Colors" section. First, the color of the bar itself can be specified under "Graph color". By default no bar background is drawn. This is the part of the bar line that is currently not filled by the bar. If the "Bar background" checkbox is ticked, the non-filled part of the bar is shown in black.

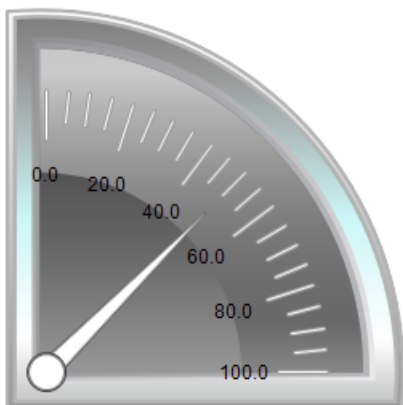
Within "Color areas" subsection, the scale can be subdivided into subareas. Each subarea can be assigned a certain color by creating a color area with the  Create new button. The color areas are numbered in ascending order. Each color area is assigned its own input fields within the element properties.

The limits of the subarea can be specified under "Begin of area" and "End of area". The color can be selected via a pull-down menu. Once a color area has been created, it can be deleted by clicking on the corresponding  Delete button.



15.8.5.2 Pointer instrument 90°

The pointer instrument can be used to add a revolution counter to the visualization, for example. The arrow positions itself based on the value of the assigned variables. The element has a preconfigured design, for which the [background_color](#) [► 546] can be set. Optional, this design can be replaced by a user-specified [background_image](#) [► 546]. The scale can be subdivided into [color areas](#) [► 548]. An example of the configuration can be found in the section "[Configuration of a pointer instrument](#) [► 549]".




Properties editor

The properties of a visualization element - except [alignment](#) and [order](#) [► 377] - can all be configured in the [properties editor](#) [► 385]. By default, this editor opens next to the visualization editor, or it can be opened explicitly via the "Properties" command (which can be found in the View menu as standard).

A property can be modified by editing the field "Value". To this end, an input field, a selection list, a dialog or checkbox that can be activated is provided in this field, depending on the element type. The value field opens

- after a double-click,
- after a single click in a selected field,
- via the space bar, if the field was already selected.

If a variable is assigned,

- simply enter its name.
- Use the  button to open the input assistant for selecting a variable. The Variables category lists all variables that have already been defined in the project.

Working in the list of properties can be made easier with the aid of default, sorting and filter functions.

Element properties

All element properties and their descriptions are listed below.

Element name	The element name can be changed. Standard name is "GenElemInst_x". "x" stands for a sequential number.
Element type	The element type is entered here. For three element groups it is possible to switch between the corresponding elements by changing the element type: <ul style="list-style-type: none"> • rectangle, rounded rectangle and ellipse [► 475] • Polygon, polyline or Bézier curve [► 490] • dip switch, power switch, push switch, push switch with LED, rocker switch [► 532]

Auxiliary setting

Value	Numeric variable, whose value is displayed as the deflection of the arrow.
--------------	--

Position

Here you can define the position (X/Y coordinates) and size (width and height) of the element in pixels. The origin is in the top left corner of the window. The positive x-axis is on the right, the positive y-axis runs downwards. If the values edited, the displayed element is simultaneously modified in the [visualization editor \[► 376\]](#).


X	Horizontal position in pixels – X=0 is the left edge of the window.
Y	Vertical position in pixels – Y=0 is the upper edge of the window.
Width	Width of the element in pixels
Height	Height of the element in pixels

Background

If no user-specified background image is used, the following properties are available:

Background color	Select a color for the bar: <ul style="list-style-type: none"> • Yellow • Red • Green • Blue • Grey
------------------	--


If a user-specified background image is used, the following properties are available:

Image	Here you can assign an image from an image pool by specifying the name of the image file or its ID.
Transparency color	For images with transparent background a color can be selected that is to be shown transparent. The  button opens the color selection dialog.



Arrow

Arrow type	A number of different arrow types are available for selection: <ul style="list-style-type: none"> • Normal arrow • Thin arrow • Wide arrow • Thin needle • Thin 3D arrow • Thin 3D needle
Color	Color, with which the arrow is displayed
Angle range	Here you can select a 90° section: <ul style="list-style-type: none"> • Top right • Top left • Bottom left • Bottom right
Additional arrow	If this option is enabled, an additional arrow is shown on the scale opposite the needle.

Scale

Sub scale position	The sub scale can be shown at the outer or inner radius of the scale ring: <ul style="list-style-type: none"> • Outside • Inside
Scale type	The scale can be shown as: <ul style="list-style-type: none"> • Lines • Points • Squares
Scale start	Lower scale limit value
Scale end	Upper scale limit value
Main scale	Distance between two lines of the coarse scale
Sub scale	Distance between two lines of the fine scale. The value can be set to 0, if further subdivision of the coarse scale is not required.
Scale line width	Width of the scale line in pixels
Scale color	The color can be set via the selection list or via the  button from the standard color selection dialog.
Scale in 3D	If this option is enabled, the scale is shown three-dimensionally.
Show scale	The scale is shown if this option is enabled.
Frame inside	If this option is enabled, the scale ring is drawn with an internal frame. This option is disabled by default.
Frame outside	If this option is enabled, the scale ring is drawn with an external frame. This option is disabled by default.

Labelling

Labelling	Here you can specify whether the scale values are positioned on the outside or the inside of the scale.
Unit	The entered text is used as element label. It is shown below the center of the scale and can be used to specify the unit of the scale, for example.
Font	Here you can set the font for the unit and the scale: <ul style="list-style-type: none"> • Standard • Heading • Large • Title • Note Click the  button to open a dialog for user-defined font property settings.
Scale format (C-Syntax)	Use the C syntax to specify the formatting of the scale label. For example, entering the string "%3.2f s" in this field results in scale labels with 3 digits, 2 of which are decimal places, followed by the letter "s".
Max. text width of labels	Value specifying the maximum width of the scale label. This value is generally set correctly automatically. Only use this setting option if the automatic adjustment does not lead to the required result.
Text height of labels	Value specifying the height of the scale label. This value is generally set correctly automatically. Only use this setting option if the automatic adjustment does not lead to the required result.
Font color	The color for the label can be set via the selection list or via the  button from the standard color selection dialog.

Positioning

Arrow distance	Arrow length in pixels
Label offset	Distance in pixels for positioning the label
Unit offset	Vertical distance in pixels for positioning the text (entered under Label → Unit)


Colors

A color is defined based on a hexadecimal number consisting of red / green / blue (RGB) components. For each of these three colors, 256 (0-255) values are available, which can be entered here statically. The color can be selected from a selection list or via the color selection dialog, which can be opened via the button



. In addition, the level of transparency can be set for each color (0: fully transparent, 255: fully opaque).

Color areas

Durable color areas	If this option is enabled, the color areas are permanently visible. The effects of this option are visible only in online mode.
[<n>]	Click on the  button to generate a new color area. For each color area an area is created that covers the corresponding settings. [<n>]: The number indexes the area. Clicking 'Delete' deletes the corresponding color area and its settings.

Area [<n>]

Begin of area	Start of the color range. It must lie within the defined scale [► 539] .
End of area	End of the color range. It must lie within the defined scale [► 539] .
Color	Color of the bar area



Transparency is not supported under Windows CE.

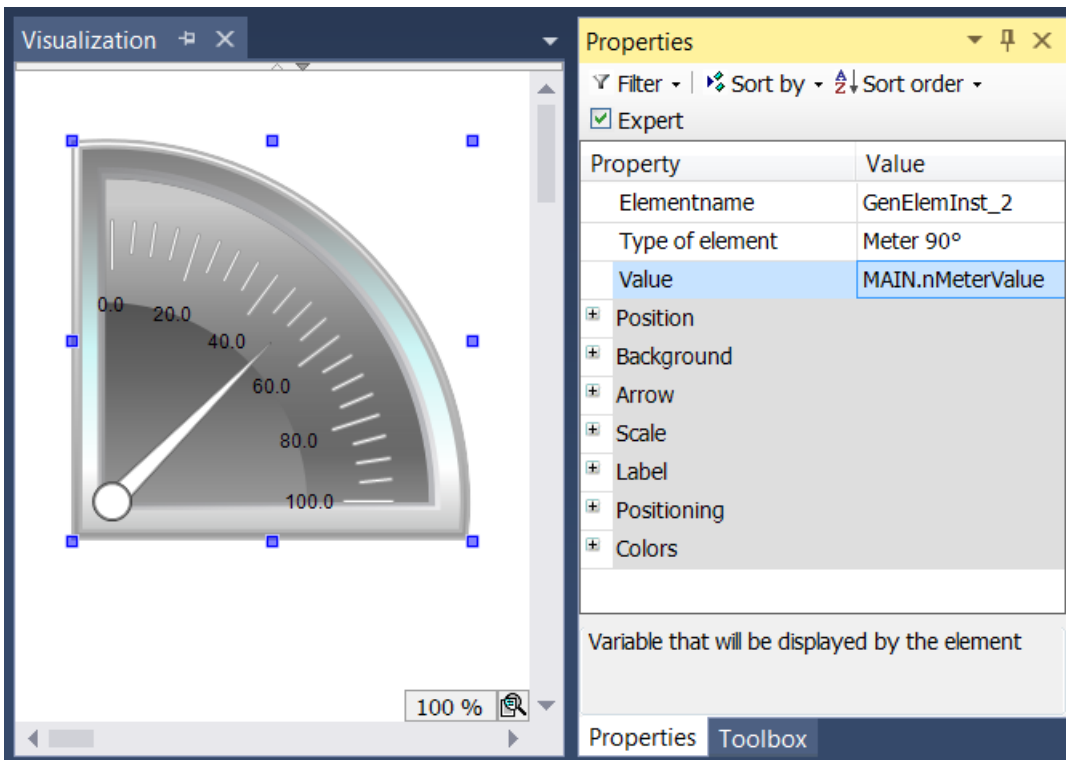
Access rights

This setting relates to the access rights for the individual element. Click to open the [Access rights dialog](#) [► 420]. The setting is only available if a [user management](#) [► 393] was added to the PLC project. The following status messages are available:

Not set. All rights.	The default message is set, if the element is shown as available for all groups.
Rights are issued: Limited rights.	The message is set, if the element is shown with limited behavior for at least one group.

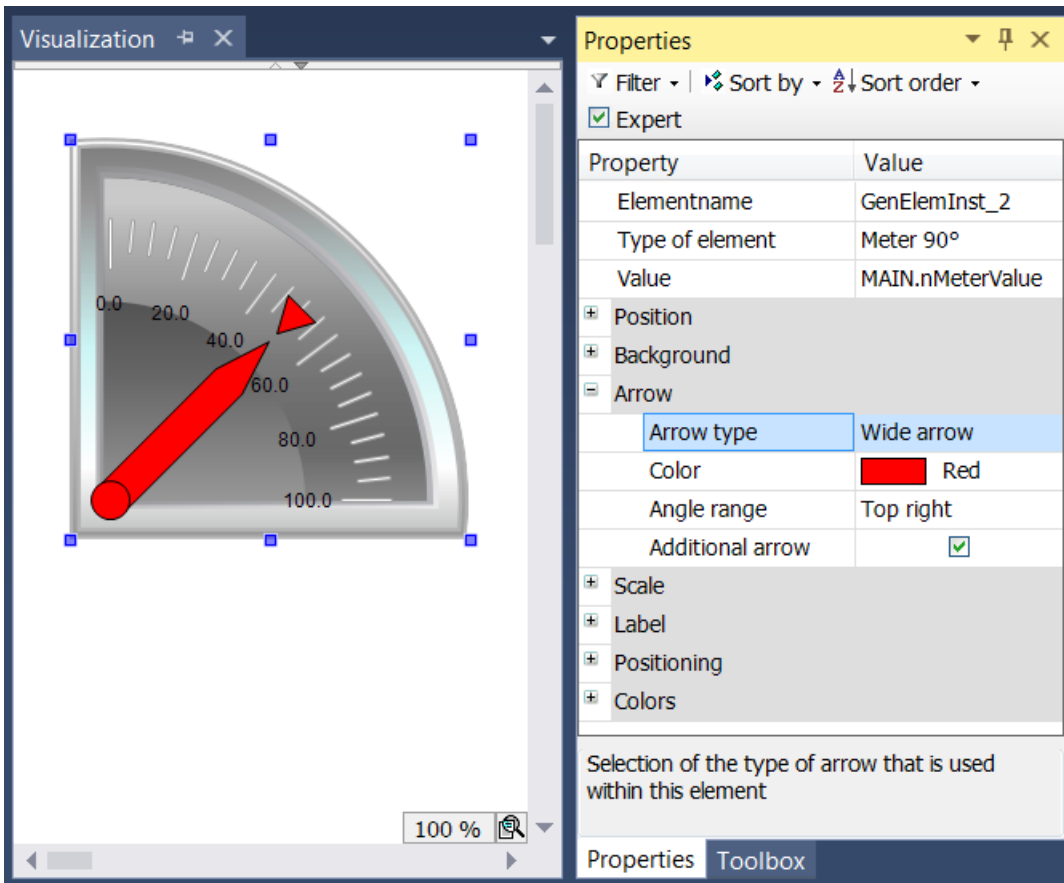
15.8.5.2.1 Configuration of a pointer instrument

The following section explains the configuration of a pointer instrument, based on an example. This example is applicable for all available pointer instruments.



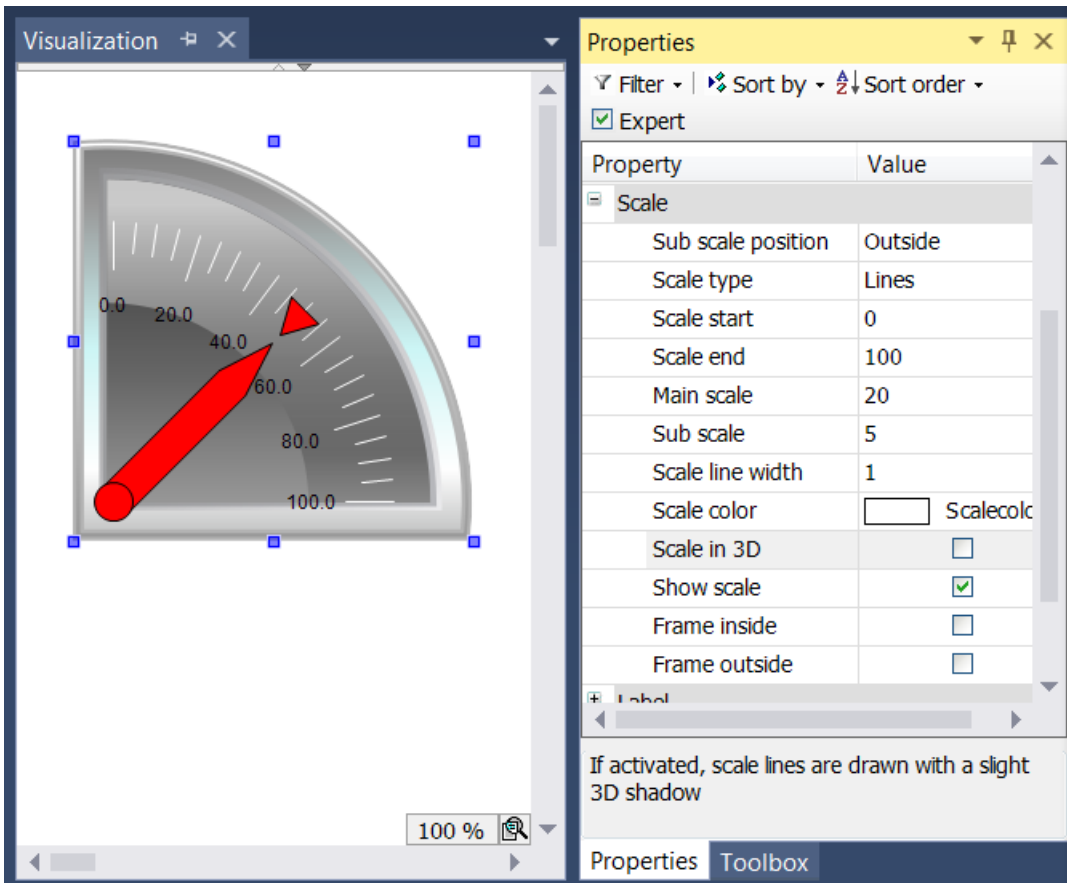
The input variable to be linked – in the example a variable with the name "nMeterValue" – should be specified under "Value" in the element properties for the pointer instrument element. The button is available after clicking in the input field. It can be used to search for the input variable within the project. Be sure to specify the input variable with its full path in the project tree.

The orientation and size of the scale display and the color and appearance of the arrow can be specified in the Arrow section of the element properties.



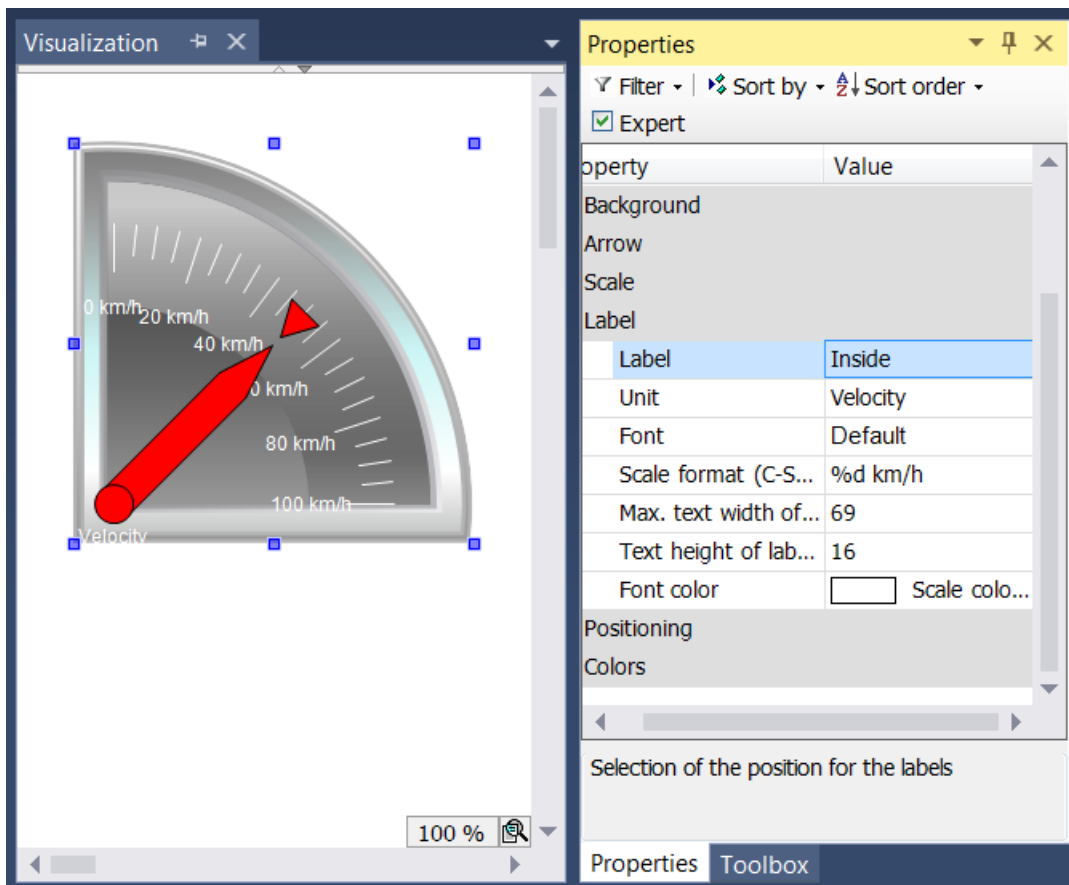
For the element with the name Pointer instrument the "Pointer start" and "Pointer end" can be set. They indicate the angle (in degrees) between the left or right edge of the scale and the horizontal line. This angle is oriented mathematically, i.e. anticlockwise, so that the value in the field "Pointer start" must always be greater than the value of "Pointer end". The pair formed by the start and end value is periodic with 360 degrees.

In the Scale section the value range for the scale (main scale and sub scale) can be specified.




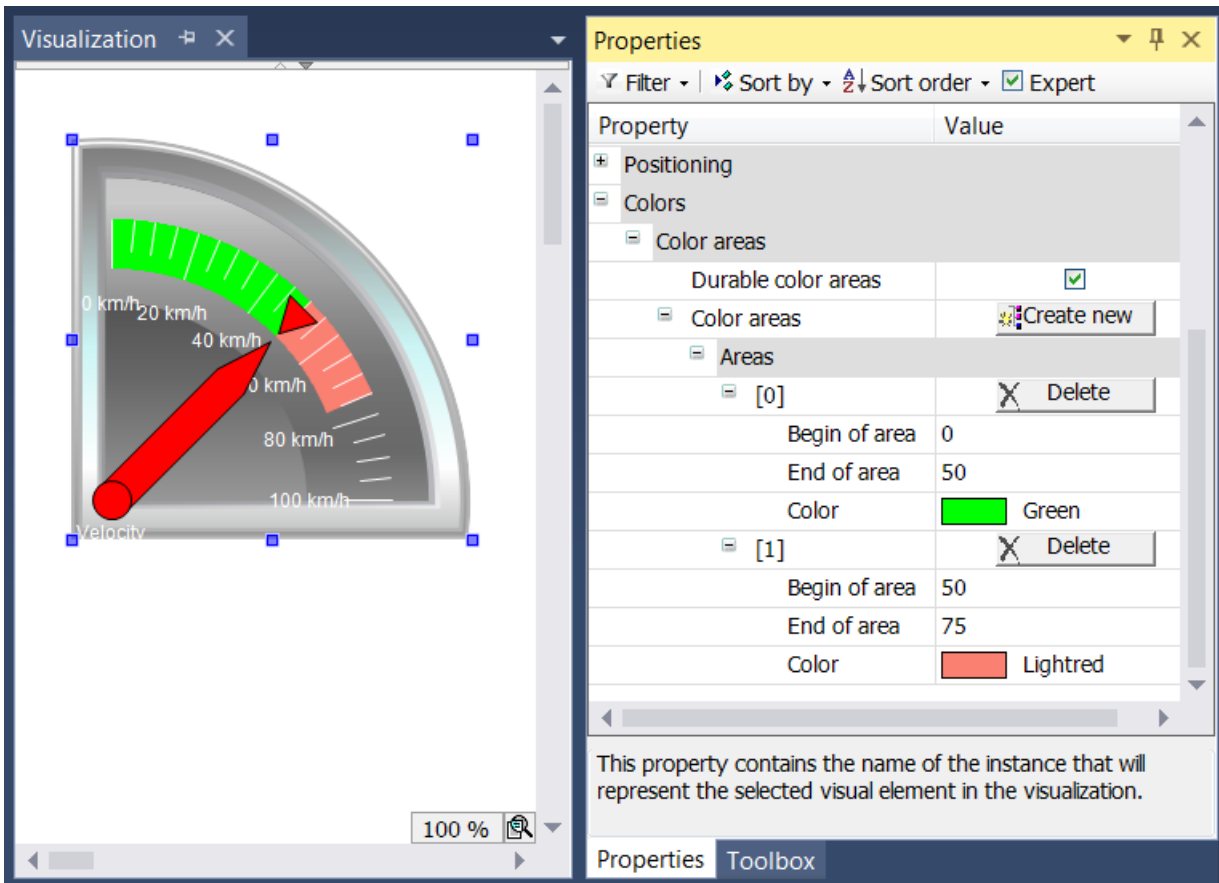
The start value for the scale is shown at the left edge of the scale. It must therefore be smaller than the end value for the scale, which is shown at the right edge of the scale. In contrast to the main scale, the fine scale (sub scale) can be omitted by setting the distance value to 0. In this case no sub scale marks would be shown. Since the checkboxes "Frame inside" and "Frame outside" were unticked in the example, the inner and outer arc of the scale are hidden.

Once the scale has been specified, the scale labels can be formatted in the "Label" section.



By changing the label from "Outside" to "Inside", the scale marks are moved to the inside of the scale. The entry in the field Unit appears below the base of the arrow. Once a suitable font and font color has been selected, the formatting of the scale marks can be adjusted. The numerical value of the scale must be formatted based on the syntax of the C programming language. Use "%d" for integer and "%.Xf" for floating-point number, whereby "X" should be replaced by the required number of decimal places. The values in the following two input fields are entered according to the settings made previously in this section. The values only have to be changed, if the automatic adjustment does not lead to the required result.

Finally, in the Colors section certain areas of the scale can be colored by creating color areas using the  Create new button. The color areas are numbered in ascending order. Each color area is assigned its own input fields within the element properties.



The fields "Begin of area" and "End of area" are available for specifying the subsection of the scale. The color can be selected from pull-down menu and deleted again with the **Delete** button.

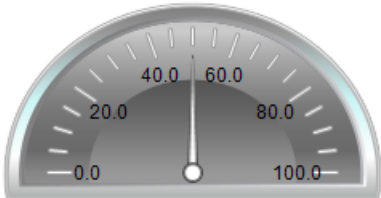
The effect of the checkbox "Durable color areas" is only apparent at runtime. In first case the checkbox is ticked, in the second case it is not ticked.



While the lower pointer instrument only shows the color area containing the arrow, the upper element shows all color areas that were created, since the checkbox "Durable color areas" was ticked.

15.8.5.3 Pointer instrument 180°

The pointer instrument can be used to add a revolution counter to the visualization, for example. The arrow positions itself based on the value of the assigned variables. The element has a preconfigured design, for which a [background color \[► 555\]](#) can be set. Optional, this design can be replaced by a user-specified [background image \[► 555\]](#). The scale can be subdivided into [color areas \[► 557\]](#). An example of the configuration can be found in the section "[Configuration of a pointer instrument \[► 557\]](#)".




Properties editor

The properties of a visualization element - except [alignment and order \[► 377\]](#) - can all be configured in the [properties editor \[► 385\]](#). By default, this editor opens next to the visualization editor, or it can be opened explicitly via the "Properties" command (which can be found in the View menu as standard).

A property can be modified by editing the field "Value". To this end, an input field, a selection list, a dialog or checkbox that can be activated is provided in this field, depending on the element type. The value field opens

- after a double-click,
- after a single click in a selected field,
- via the space bar, if the field was already selected.

If a variable is assigned,

- simply enter its name.
- Use the  button to open the input assistant for selecting a variable. The Variables category lists all variables that have already been defined in the project.

Working in the list of properties can be made easier with the aid of default, sorting and filter functions.

Element properties

All element properties and their descriptions are listed below.

Element name	The element name can be changed. Standard name is "GenElemInst_x". "x" stands for a sequential number.
Element type	The element type is entered here. For three element groups it is possible to switch between the corresponding elements by changing the element type: <ul style="list-style-type: none"> • rectangle, rounded rectangle and ellipse [► 475] • Polygon, polyline or Bézier curve [► 490] • dip switch, power switch, push switch, push switch with LED, rocker switch [► 532]

Auxiliary setting

Value	Numeric variable, whose value is displayed as the deflection of the arrow.
--------------	--

Position

Here you can define the position (X/Y coordinates) and size (width and height) of the element in pixels. The origin is in the top left corner of the window. The positive x-axis is on the right, the positive y-axis runs downwards. If the values edited, the displayed element is simultaneously modified in the [visualization editor](#) [► 376].


X	Horizontal position in pixels – X=0 is the left edge of the window.
Y	Vertical position in pixels – Y=0 is the upper edge of the window.
Width	Width of the element in pixels
Height	Height of the element in pixels

Background

If no user-specified background image is used, the following properties are available:

Background color	Select a color for the bar: <ul style="list-style-type: none"> • Yellow • Red • Green • Blue • Grey
------------------	--


If a user-specified background image is used, the following properties are available:

Image	Here you can assign an image from an image pool by specifying the name of the image file or its ID.
Transparency color	For images with transparent background a color can be selected that is to be shown transparent. The  button opens the color selection dialog.



Arrow

Arrow type	A number of different arrow types are available for selection: <ul style="list-style-type: none"> • Normal arrow • Thin arrow • Wide arrow • Thin needle • Thin 3D arrow • Thin 3D needle
Color	Color, with which the arrow is displayed
Angle range	Here you can select a 180° section: <ul style="list-style-type: none"> • Top • Bottom • Left • Right
Additional arrow	If this option is enabled, an additional arrow is shown on the scale opposite the needle.

Scale

Sub scale position	The sub scale can be shown at the outer or inner radius of the scale ring: <ul style="list-style-type: none"> • Outside • Inside
Scale type	The scale can be shown as: <ul style="list-style-type: none"> • Lines • Points • Squares
Scale start	Lower scale limit value
Scale end	Upper scale limit value
Main scale	Distance between two lines of the coarse scale
Sub scale	Distance between two lines of the fine scale. The value can be set to 0, if further subdivision of the coarse scale is not required.
Scale line width	Width of the scale line in pixels
Scale color	The color can be set via the selection list or via the  button from the standard color selection dialog.
Scale in 3D	If this option is enabled, the scale is shown three-dimensionally.
Show scale	The scale is shown if this option is enabled.
Frame inside	If this option is enabled, the scale ring is drawn with an internal frame. This option is disabled by default.
Frame outside	If this option is enabled, the scale ring is drawn with an external frame. This option is disabled by default.

Labelling

Labelling	Here you can specify whether the scale values are positioned on the outside or the inside of the scale.
Unit	The entered text is used as element label. It is shown below the center of the scale and can be used to specify the unit of the scale, for example.
Font	Here you can set the font for the unit and the scale: <ul style="list-style-type: none"> • Standard • Heading • Large • Title • Note <p>Click the  button to open a dialog for user-defined font property settings.</p>
Scale format (C-Syntax)	Use the C syntax to specify the formatting of the scale label. For example, entering the string "%3.2f s" in this field results in scale labels with 3 digits, 2 of which are decimal places, followed by the letter "s".
Max. text width of labels	Value specifying the maximum width of the scale label. This value is generally set correctly automatically. Only use this setting option if the automatic adjustment does not lead to the required result.
Text height of labels	Value specifying the height of the scale label. This value is generally set correctly automatically. Only use this setting option if the automatic adjustment does not lead to the required result.
Font color	The color for the label can be set via the selection list or via the  button from the standard color selection dialog.

Positioning

Arrow distance	Arrow length in pixels
Label offset	Distance in pixels for positioning the label
Unit offset	Vertical distance in pixels for positioning the text (entered under Label → Unit)

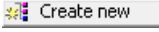
Colors

A color is defined based on a hexadecimal number consisting of red / green / blue (RGB) components. For each of these three colors, 256 (0-255) values are available, which can be entered here statically. The color can be selected from a selection list or via the color selection dialog, which can be opened via the button



. In addition, the level of transparency can be set for each color (0: fully transparent, 255: fully opaque).

Color areas

Durable color areas	If this option is enabled, the color areas are permanently visible. The effects of this option are visible only in online mode.
[<n>]	Click on the  Create new button to generate a new color area. For each color area an area is created that covers the corresponding settings. [<n>]: The number indexes the area. Clicking 'Delete' deletes the corresponding color area and its settings.

Area [<n>]

Begin of area	Start of the color range. It must lie within the defined <u>scale</u> [▶ 539].
End of area	End of the color range. It must lie within the defined <u>scale</u> [▶ 539].
Color	Color of the bar area



Transparency is not supported under Windows CE.

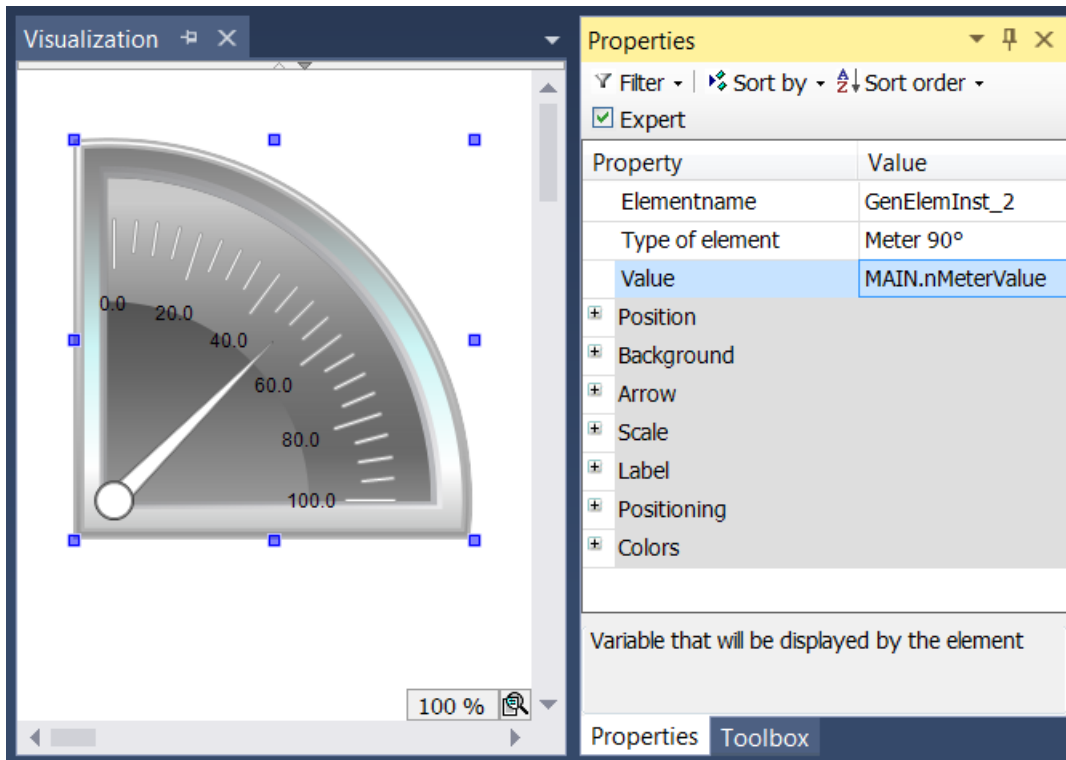
Access rights


This setting relates to the access rights for the individual element. Click to open the Access rights dialog [▶ 420]. The setting is only available if a user management [▶ 393] was added to the PLC project. The following status messages are available:

Not set. All rights.	The default message is set, if the element is shown as available for all groups.
Rights are issued: Limited rights.	The message is set, if the element is shown with limited behavior for at least one group.

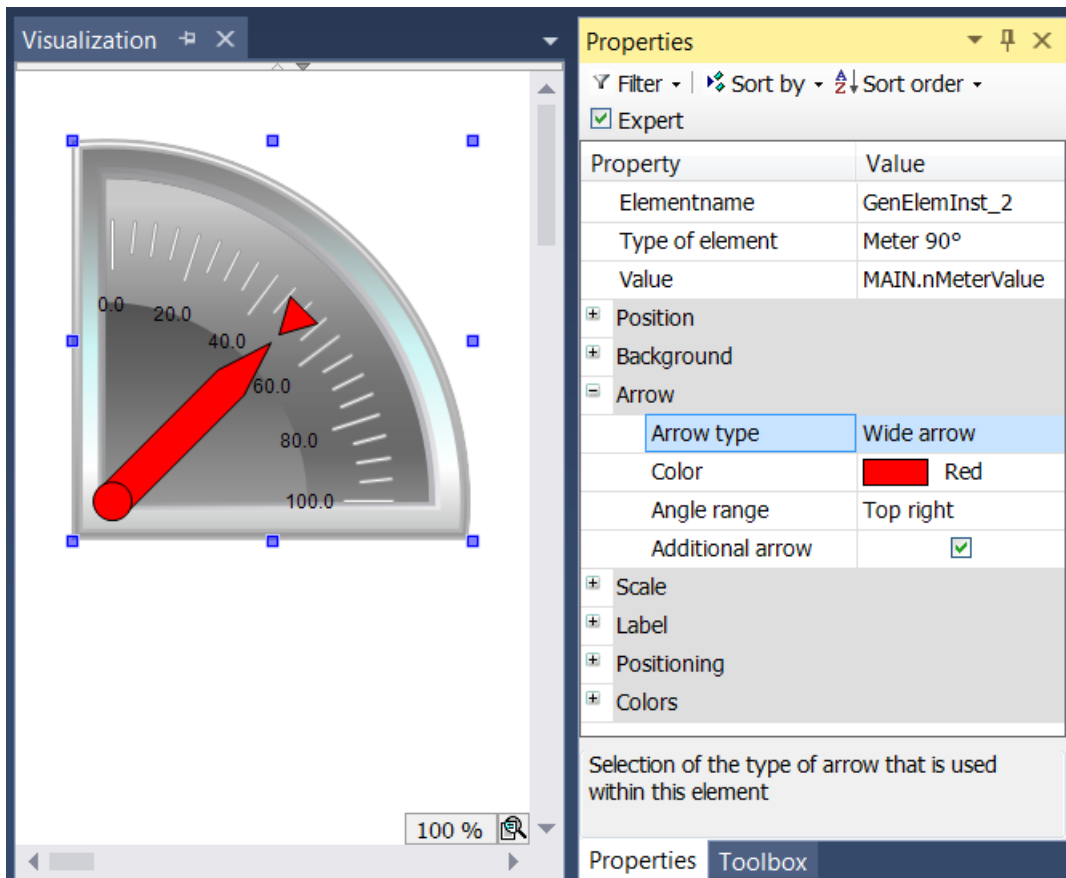
15.8.5.3.1 Configuration of a pointer instrument

The following section explains the configuration of a pointer instrument, based on an example. This example is applicable for all available pointer instruments.



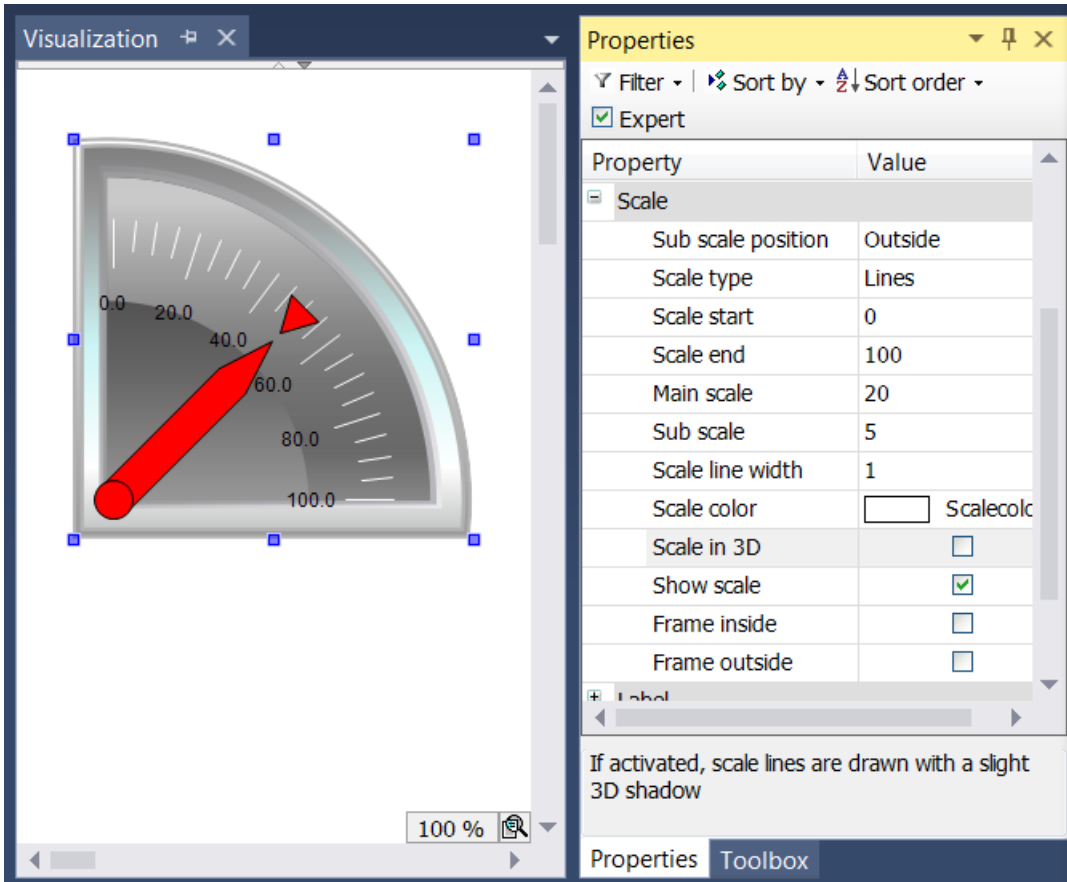
The input variable to be linked – in the example a variable with the name "nMeterValue" – should be specified under "Value" in the element properties for the pointer instrument element. The  button is available after clicking in the input field. It can be used to search for the input variable within the project. Be sure to specify the input variable with its full path in the project tree.

The orientation and size of the scale display and the color and appearance of the arrow can be specified in the Arrow section of the element properties.



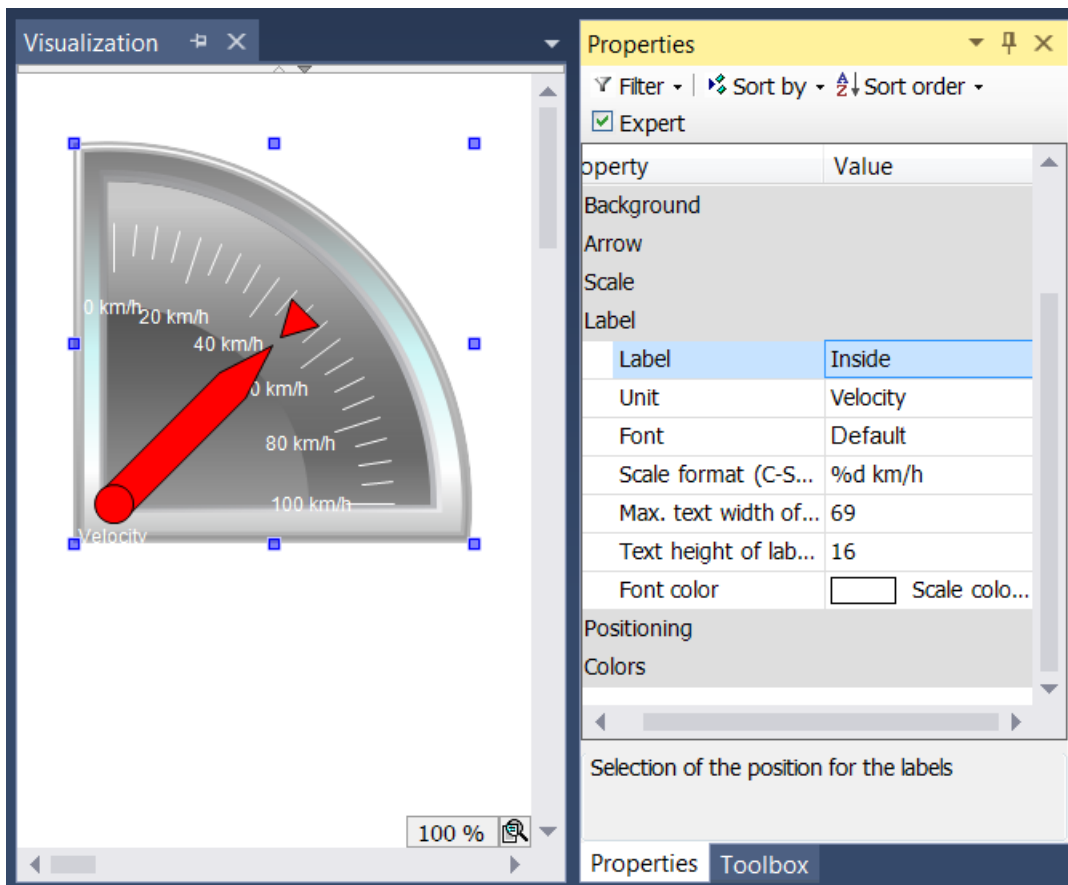
For the element with the name Pointer instrument the "Pointer start" and "Pointer end" can be set. They indicate the angle (in degrees) between the left or right edge of the scale and the horizontal line. This angle is oriented mathematically, i.e. anticlockwise, so that the value in the field "Pointer start" must always be greater than the value of "Pointer end". The pair formed by the start and end value is periodic with 360 degrees.

In the Scale section the value range for the scale (main scale and sub scale) can be specified.




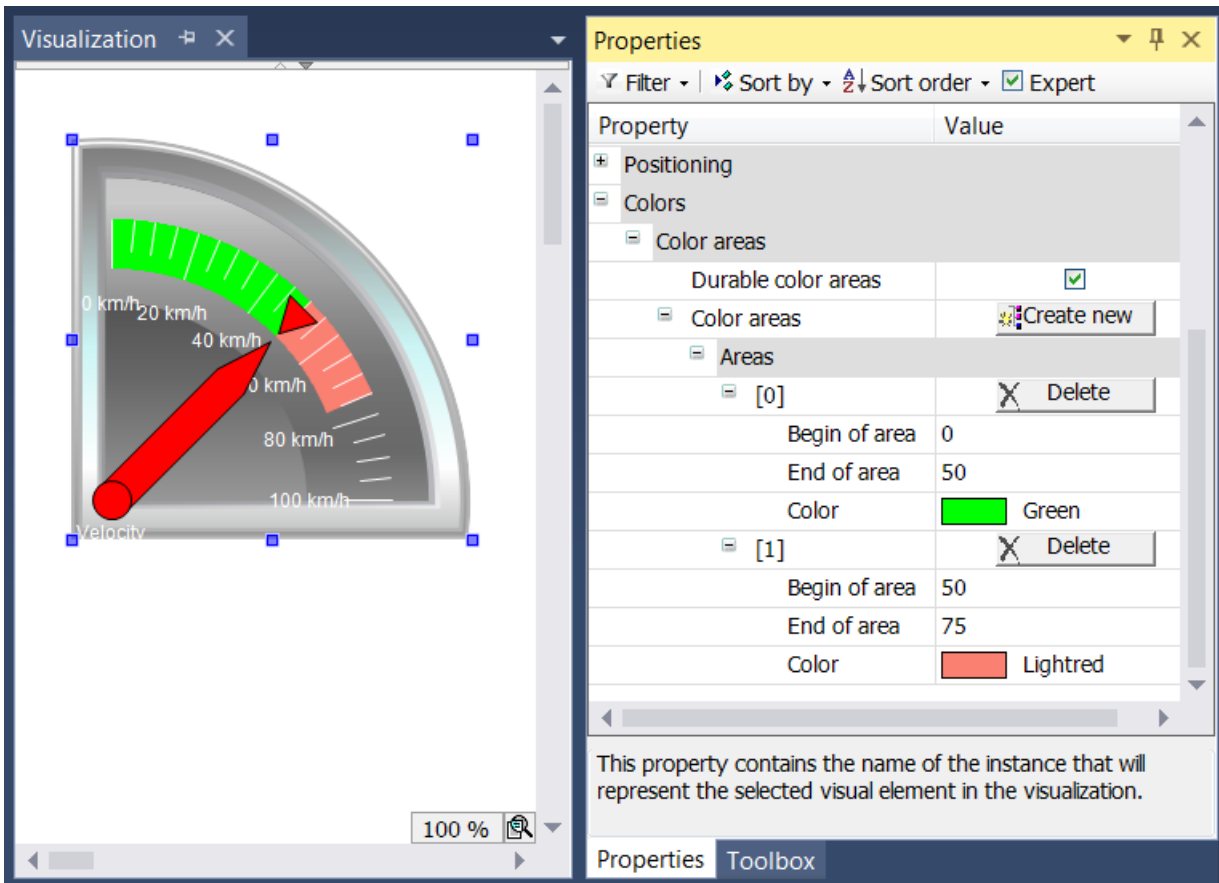
The start value for the scale is shown at the left edge of the scale. It must therefore be smaller than the end value for the scale, which is shown at the right edge of the scale. In contrast to the main scale, the fine scale (sub scale) can be omitted by setting the distance value to 0. In this case no sub scale marks would be shown. Since the checkboxes "Frame inside" and "Frame outside" were unticked in the example, the inner and outer arc of the scale are hidden.

Once the scale has been specified, the scale labels can be formatted in the "Label" section.



By changing the label from "Outside" to "Inside", the scale marks are moved to the inside of the scale. The entry in the field Unit appears below the base of the arrow. Once a suitable font and font color has been selected, the formatting of the scale marks can be adjusted. The numerical value of the scale must be formatted based on the syntax of the C programming language. Use "%d" for integer and "%.Xf" for floating-point number, whereby "X" should be replaced by the required number of decimal places. The values in the following two input fields are entered according to the settings made previously in this section. The values only have to be changed, if the automatic adjustment does not lead to the required result.

Finally, in the Colors section certain areas of the scale can be colored by creating color areas using the  Create new button. The color areas are numbered in ascending order. Each color area is assigned its own input fields within the element properties.



The fields "Begin of area" and "End of area" are available for specifying the subsection of the scale. The color can be selected from pull-down menu and deleted again with the button.

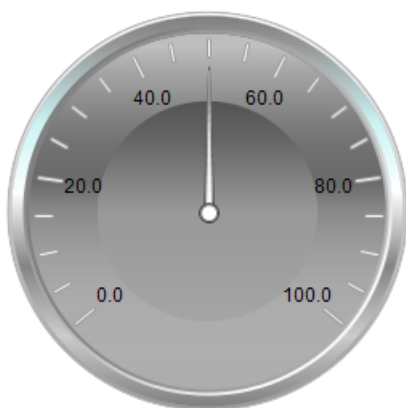
The effect of the checkbox "Durable color areas" is only apparent at runtime. In first case the checkbox is ticked, in the second case it is not ticked.



While the lower pointer instrument only shows the color area containing the arrow, the upper element shows all color areas that were created, since the checkbox "Durable color areas" was ticked.

15.8.5.4 Pointer instrument

The pointer instrument can be used to add a revolution counter to the visualization, for example. The arrow positions itself based on the value of the assigned variables. The element has a preconfigured design, for which the [background color \[► 563\]](#) can be set. Optional, this design can be replaced by a user-specified [background image \[► 563\]](#). The scale can be subdivided into [color areas \[► 565\]](#). An example of the configuration can be found in the section "[Configuration of a pointer instrument \[► 565\]](#)".



Properties editor

The properties of a visualization element - except [alignment and order \[► 377\]](#) - can all be configured in the [properties editor \[► 385\]](#). By default, this editor opens next to the visualization editor, or it can be opened explicitly via the "Properties" command (which can be found in the View menu as standard).

A property can be modified by editing the field "Value". To this end, an input field, a selection list, a dialog or checkbox that can be activated is provided in this field, depending on the element type. The value field opens

- after a double-click,
- after a single click in a selected field,
- via the space bar, if the field was already selected.

If a variable is assigned,

- simply enter its name.
- Use the button to open the input assistant for selecting a variable. The Variables category lists all variables that have already been defined in the project.

Working in the list of properties can be made easier with the aid of default, sorting and filter functions.

Element properties

All element properties and their descriptions are listed below.

Element name	The element name can be changed. Standard name is "GenElemInst_x". "x" stands for a sequential number.
Element type	The element type is entered here. For three element groups it is possible to switch between the corresponding elements by changing the element type: <ul style="list-style-type: none"> • rectangle, rounded rectangle and ellipse [► 475] • Polygon, polyline or Bézier curve [► 490] • dip switch, power switch, push switch, push switch with LED, rocker switch [► 532]

Auxiliary setting

Value	Numeric variable, whose value is displayed as the deflection of the arrow.
--------------	--

Position

Here you can define the position (X/Y coordinates) and size (width and height) of the element in pixels. The origin is in the top left corner of the window. The positive x-axis is on the right, the positive y-axis runs downwards. If the values edited, the displayed element is simultaneously modified in the [visualization editor](#) [▶ 376].


X	Horizontal position in pixels – X=0 is the left edge of the window.
Y	Vertical position in pixels – Y=0 is the upper edge of the window.
Width	Width of the element in pixels
Height	Height of the element in pixels

Background

If no user-specified background image is used, the following properties are available:

Background color	Select a color for the bar: <ul style="list-style-type: none"> • Yellow • Red • Green • Blue • Grey
------------------	--


If a user-specified background image is used, the following properties are available:

Image	Here you can assign an image from an image pool by specifying the name of the image file or its ID.
Transparency color	For images with transparent background a color can be selected that is to be shown transparent. The  button opens the color selection dialog.



Arrow

Arrow type	A number of different arrow types are available for selection: <ul style="list-style-type: none"> • Normal arrow • Thin arrow • Wide arrow • Thin needle • Thin 3D arrow • Thin 3D needle
Color	Color, with which the arrow is displayed
Pointer start	The value specified here is interpreted as an angle in degrees, which ranges from the origin of the scale to the start of the scale area in anticlockwise direction. The origin of the scale is at "3 o'clock". The angle has a value range between 0° and 360°. The center of the rotation is the X/Y position.
Pointer end	The value specified here is interpreted as an angle in degrees, which ranges from the origin of the scale to the end of the scale area in anticlockwise direction. The origin of the scale is at "3 o'clock". Negative values are permitted. The center of the rotation is the X/Y position. The pointer end must be set such that it is to the right of the pointer start, viewed in clockwise direction. The angle has a value range between 0° and a maximum of 350°.
Additional arrow	If this option is enabled, an additional arrow is shown on the scale opposite the needle.

Scale

Sub scale position	The sub scale can be shown at the outer or inner radius of the scale ring: <ul style="list-style-type: none"> • Outside • Inside
Scale type	The scale can be shown as: <ul style="list-style-type: none"> • Lines • Points • Squares
Scale start	Lower scale limit value
Scale end	Upper scale limit value
Main scale	Distance between two lines of the coarse scale
Sub scale	Distance between two lines of the fine scale. The value can be set to 0, if further subdivision of the coarse scale is not required.
Scale line width	Width of the scale line in pixels
Scale color	The color can be set via the selection list or via the  button from the standard color selection dialog.
Scale in 3D	If this option is enabled, the scale is shown three-dimensionally.
Show scale	The scale is shown if this option is enabled.
Frame inside	If this option is enabled, the scale ring is drawn with an internal frame. This option is disabled by default.
Frame outside	If this option is enabled, the scale ring is drawn with an external frame. This option is disabled by default.

Labelling

Labelling	Here you can specify whether the scale values are positioned on the outside or the inside of the scale.
Unit	The entered text is used as element label. It is shown below the center of the scale and can be used to specify the unit of the scale, for example.
Font	Here you can set the font for the unit and the scale: <ul style="list-style-type: none"> • Standard • Heading • Large • Title • Note Click the  button to open a dialog for user-defined font property settings.
Scale format (C-Syntax)	Use the C syntax to specify the formatting of the scale label. For example, entering the string "%3.2f s" in this field results in scale labels with 3 digits, 2 of which are decimal places, followed by the letter "s".
Max. text width of labels	Value specifying the maximum width of the scale label. This value is generally set correctly automatically. Only use this setting option if the automatic adjustment does not lead to the required result.
Text height of labels	Value specifying the height of the scale label. This value is generally set correctly automatically. Only use this setting option if the automatic adjustment does not lead to the required result.
Font color	The color for the label can be set via the selection list or via the  button from the standard color selection dialog.

Positioning

Arrow distance	Arrow length in pixels
Scale distance	Distance from the scale lines to the center in pixels. This property is available only if you have defined your own background image.
Scale length	Length of the scale lines in pixels. This property is available only if you have defined your own background image.
Label offset	Distance in pixels for positioning the label
Unit offset	Vertical distance in pixels for positioning the text (entered under Label → Unit)
Origin offset	Offset of the element. It can be used to achieve exact positioning in relation to the background image.


Colors

A color is defined based on a hexadecimal number consisting of red / green / blue (RGB) components. For each of these three colors, 256 (0-255) values are available, which can be entered here statically. The color can be selected from a selection list or via the color selection dialog, which can be opened via the button



. In addition, the level of transparency can be set for each color (0: fully transparent, 255: fully opaque).

Color areas

Durable color areas	If this option is enabled, the color areas are permanently visible. The effects of this option are visible only in online mode.
[<n>]	Click on the  Create new button to generate a new color area. For each color area an area is created that covers the corresponding settings. [<n>]: The number indexes the area. Clicking 'Delete' deletes the corresponding color area and its settings.

Area [<n>]

Begin of area	Start of the color range. It must lie within the defined scale [▶ 539].
End of area	End of the color range. It must lie within the defined scale [▶ 539].
Color	Color of the bar area



Transparency is not supported under Windows CE.

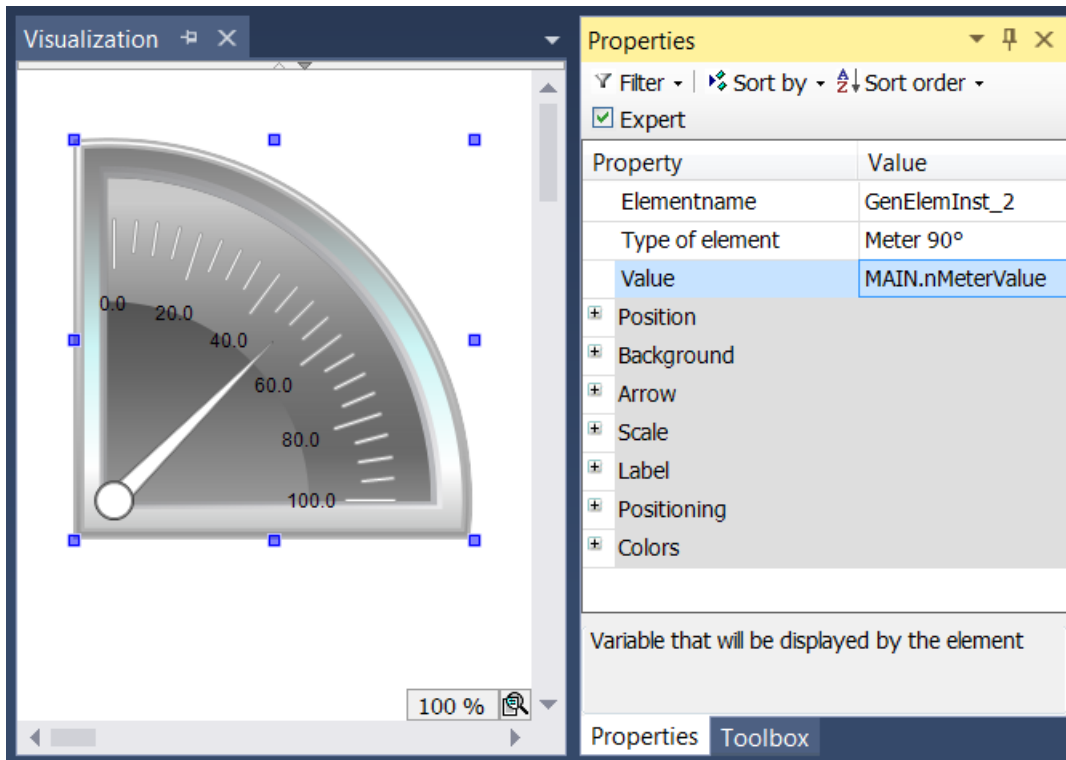
Access rights


This setting relates to the access rights for the individual element. Click to open the [Access rights dialog](#) [▶ 420]. The setting is only available if a [user management](#) [▶ 393] was added to the PLC project. The following status messages are available:

Not set. All rights.	The default message is set, if the element is shown as available for all groups.
Rights are issued: Limited rights.	The message is set, if the element is shown with limited behavior for at least one group.

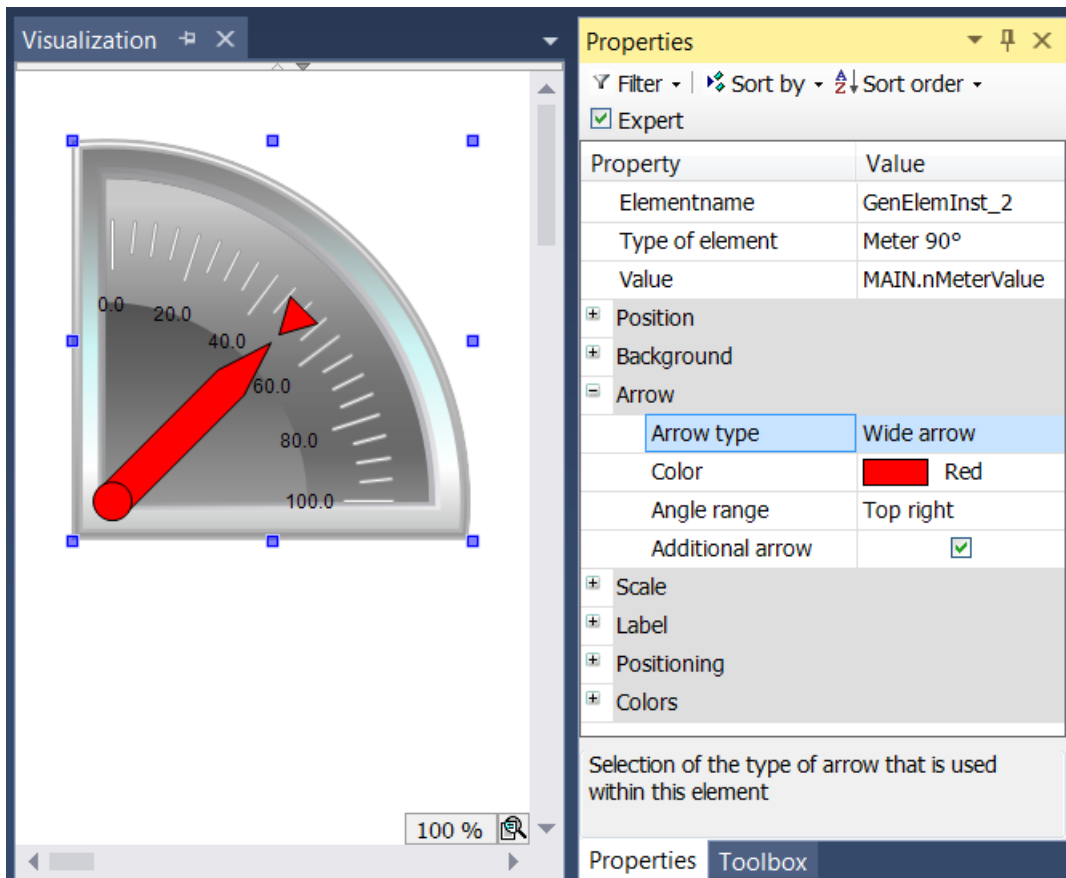
15.8.5.4.1 Configuration of a pointer instrument

The following section explains the configuration of a pointer instrument, based on an example. This example is applicable for all available pointer instruments.



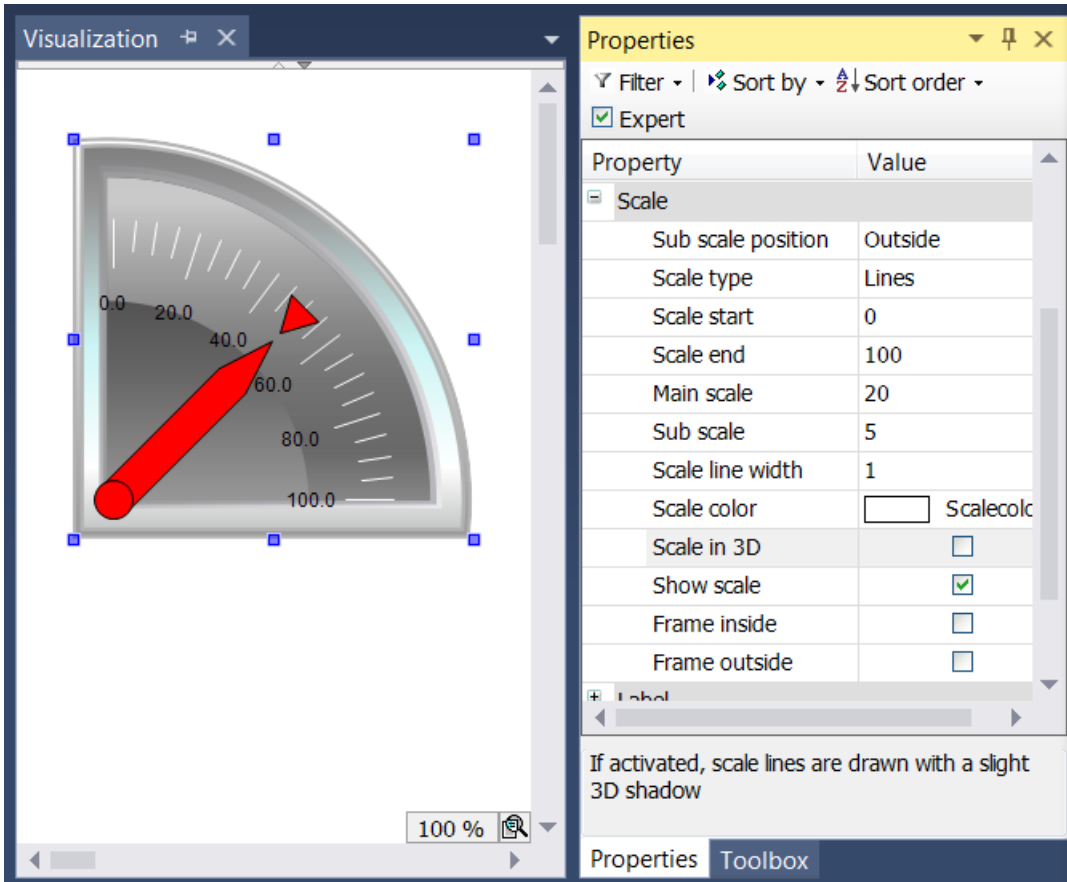
The input variable to be linked – in the example a variable with the name "nMeterValue" – should be specified under "Value" in the element properties for the pointer instrument element. The  button is available after clicking in the input field. It can be used to search for the input variable within the project. Be sure to specify the input variable with its full path in the project tree.

The orientation and size of the scale display and the color and appearance of the arrow can be specified in the Arrow section of the element properties.



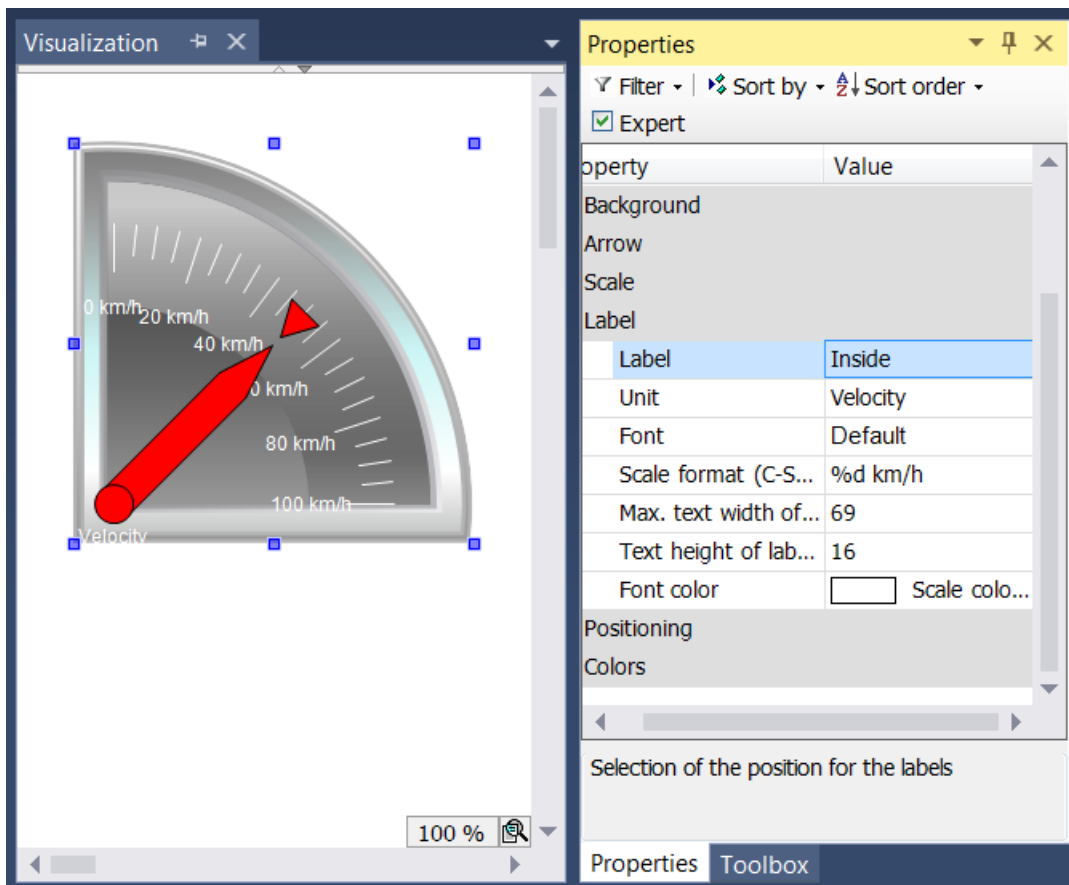
For the element with the name Pointer instrument the "Pointer start" and "Pointer end" can be set. They indicate the angle (in degrees) between the left or right edge of the scale and the horizontal line. This angle is oriented mathematically, i.e. anticlockwise, so that the value in the field "Pointer start" must always be greater than the value of "Pointer end". The pair formed by the start and end value is periodic with 360 degrees.

In the Scale section the value range for the scale (main scale and sub scale) can be specified.




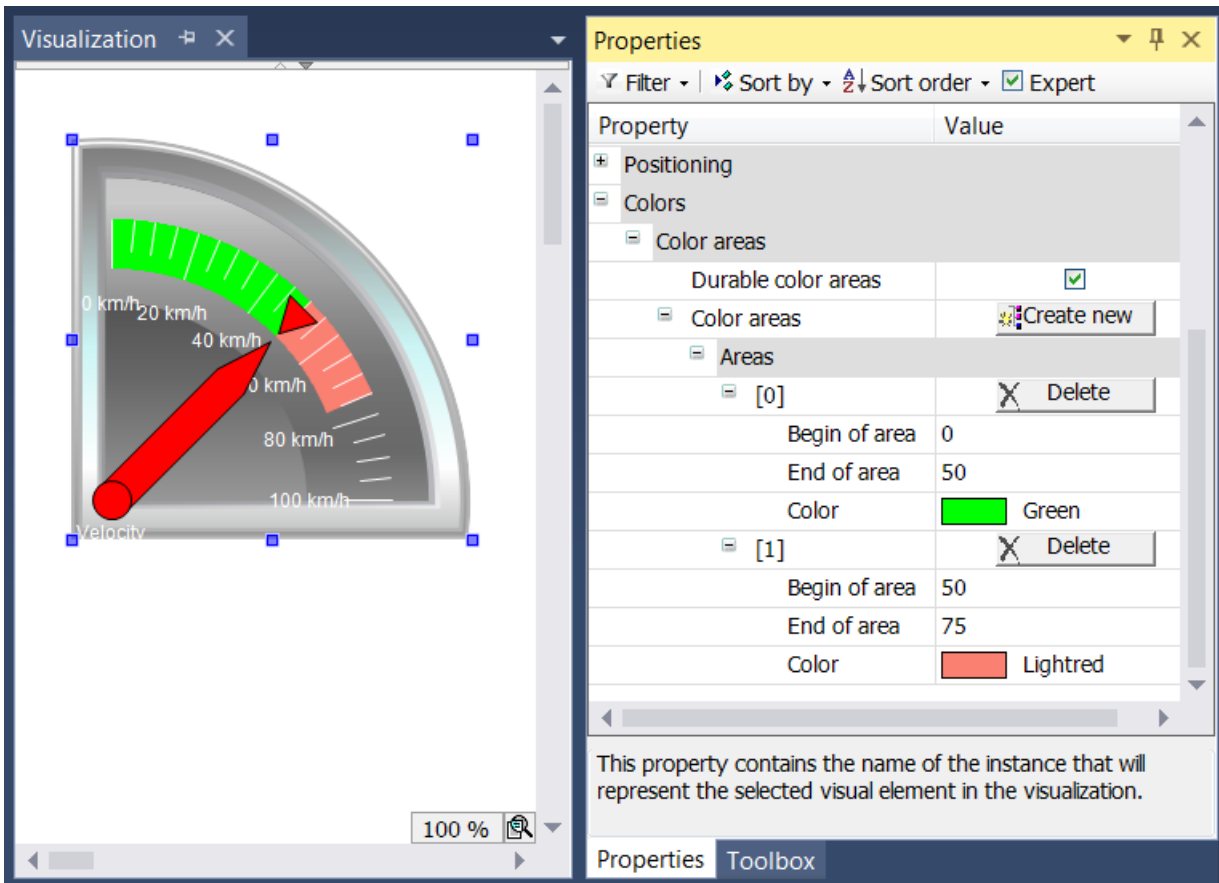
The start value for the scale is shown at the left edge of the scale. It must therefore be smaller than the end value for the scale, which is shown at the right edge of the scale. In contrast to the main scale, the fine scale (sub scale) can be omitted by setting the distance value to 0. In this case no sub scale marks would be shown. Since the checkboxes "Frame inside" and "Frame outside" were unticked in the example, the inner and outer arc of the scale are hidden.


Once the scale has been specified, the scale labels can be formatted in the "Label" section.



By changing the label from "Outside" to "Inside", the scale marks are moved to the inside of the scale. The entry in the field Unit appears below the base of the arrow. Once a suitable font and font color has been selected, the formatting of the scale marks can be adjusted. The numerical value of the scale must be formatted based on the syntax of the C programming language. Use "%d" for integer and "%.Xf" for floating-point number, whereby "X" should be replaced by the required number of decimal places. The values in the following two input fields are entered according to the settings made previously in this section. The values only have to be changed, if the automatic adjustment does not lead to the required result.

Finally, in the Colors section certain areas of the scale can be colored by creating color areas using the  Create new button. The color areas are numbered in ascending order. Each color area is assigned its own input fields within the element properties.



The fields "Begin of area" and "End of area" are available for specifying the subsection of the scale. The color can be selected from pull-down menu and deleted again with the  button.

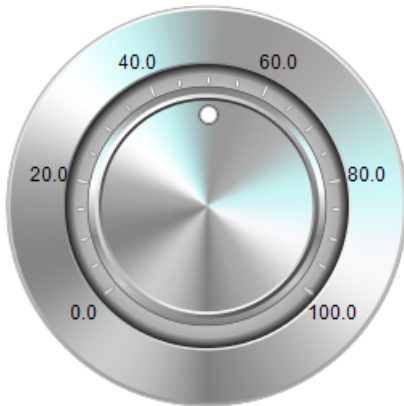
The effect of the checkbox "Durable color areas" is only apparent at runtime. In first case the checkbox is ticked, in the second case it is not ticked.



While the lower pointer instrument only shows the color area containing the arrow, the upper element shows all color areas that were created, since the checkbox "Durable color areas" was ticked.

15.8.5.5 Potentiometer

This element can be used to add a potentiometer with predefined design to a visualization page. Such an operating element uses an arrow or pointer to indicate a variable value that can be moved via the user input.



Properties editor

The properties of a visualization element - except alignment and order [[▶ 377](#)] - can all be configured in the properties editor [[▶ 385](#)]. By default, this editor opens next to the visualization editor, or it can be opened explicitly via the "Properties" command (which can be found in the View menu as standard).

A property can be modified by editing the field "Value". To this end, an input field, a selection list, a dialog or checkbox that can be activated is provided in this field, depending on the element type. The value field opens

- after a double-click,
- after a single click in a selected field,
- via the space bar, if the field was already selected.

If a variable is assigned,

- simply enter its name.
- Use the button to open the input assistant for selecting a variable. The Variables category lists all variables that have already been defined in the project.

Working in the list of properties can be made easier with the aid of default, sorting and filter functions.

Element properties

All element properties and their descriptions are listed below.

Element name	The element name can be changed. Standard name is "GenElemInst_x". "x" stands for a sequential number.
Element type	The element type is entered here. For three element groups it is possible to switch between the corresponding elements by changing the element type: <ul style="list-style-type: none"> • rectangle, rounded rectangle and ellipse [▶ 475] • Polygon, polyline or Bézier curve [▶ 490] • dip switch, power switch, push switch, push switch with LED, rocker switch [▶ 532]

Auxiliary setting

Variable	Numeric variable containing the position of the potentiometer.
-----------------	--

Position

Here you can define the position (X/Y coordinates) and size (width and height) of the element in pixels. The origin is in the top left corner of the window. The positive x-axis is on the right, the positive y-axis runs downwards. If the values edited, the displayed element is simultaneously modified in the [visualization editor](#) [► 376].


X	Horizontal position in pixels – X=0 is the left edge of the window.
Y	Vertical position in pixels – Y=0 is the upper edge of the window.
Width	Width of the element in pixels
Height	Height of the element in pixels

Background

If no user-specified background image is used, the following properties are available:

Background color	Select a color for the bar: <ul style="list-style-type: none"> • Yellow • Red • Green • Blue • Grey
------------------	--


If a user-specified background image is used, the following properties are available:

Image	Here you can assign an image from an image pool by specifying the name of the image file or its ID.
Transparency color	For images with transparent background a color can be selected that is to be shown transparent. The  button opens the color selection dialog.



Arrow

Arrow type	A number of different arrow types are available for selection: <ul style="list-style-type: none"> • Normal arrow • Thin arrow • Wide arrow • Thin needle • Thin 3D arrow • Thin 3D needle
Color	Color, with which the arrow is displayed
Pointer start	The value specified here is interpreted as an angle in degrees, which ranges from the origin of the scale to the start of the scale area in anticlockwise direction. The origin of the scale is at "3 o'clock". The angle has a value range between 0° and 360°. The center of the rotation is the X/Y position.
Pointer end	The value specified here is interpreted as an angle in degrees, which ranges from the origin of the scale to the end of the scale area in anticlockwise direction. The origin of the scale is at "3 o'clock". Negative values are permitted. The center of the rotation is the X/Y position. The pointer end must be set such that it is to the right of the pointer start, viewed in clockwise direction. The angle has a value range between 0° and a maximum of 350°.

Scale

Sub scale position	The sub scale can be shown at the outer or inner radius of the scale ring: <ul style="list-style-type: none"> • Outside • Inside
Scale type	The scale can be shown as: <ul style="list-style-type: none"> • Lines • Points • Squares
Scale start	Lower scale limit value
Scale end	Upper scale limit value
Main scale	Distance between two lines of the coarse scale
Sub scale	Distance between two lines of the fine scale. The value can be set to 0, if further subdivision of the coarse scale is not required.
Scale line width	Width of the scale line in pixels
Scale color	The color can be set via the selection list or via the  button from the standard color selection dialog.
Scale in 3D	If this option is enabled, the scale is shown three-dimensionally.
Show scale	The scale is shown if this option is enabled.
Frame inside	If this option is enabled, the scale ring is drawn with an internal frame. This option is disabled by default.
Frame outside	If this option is enabled, the scale ring is drawn with an external frame. This option is disabled by default.

Labelling

Labelling	Here you can specify whether the scale values are positioned on the outside or the inside of the scale.
Unit	The entered text is used as element label. It is shown below the center of the scale and can be used to specify the unit of the scale, for example.
Font	Here you can set the font for the unit and the scale: <ul style="list-style-type: none"> • Standard • Heading • Large • Title • Note Click the  button to open a dialog for user-defined font property settings.
Scale format (C-Syntax)	Use the C syntax to specify the formatting of the scale label. For example, entering the string "%3.2f s" in this field results in scale labels with 3 digits, 2 of which are decimal places, followed by the letter "s".
Max. text width of labels	Value specifying the maximum width of the scale label. This value is generally set correctly automatically. Only use this setting option if the automatic adjustment does not lead to the required result.
Text height of labels	Value specifying the height of the scale label. This value is generally set correctly automatically. Only use this setting option if the automatic adjustment does not lead to the required result.
Font color	The color for the label can be set via the selection list or via the  button from the standard color selection dialog.

Positioning

Arrow distance	Arrow length in pixels
Scale distance	Distance from the scale lines to the center in pixels. This property is available only if you have defined your own background image.
Scale length	Length of the scale lines in pixels. This property is available only if you have defined your own background image.
Label offset	Distance in pixels for positioning the label
Unit offset	Vertical distance in pixels for positioning the text (entered under Label → Unit)
Origin offset	Offset of the element. It can be used to achieve exact positioning in relation to the background image.


Colors

A color is defined based on a hexadecimal number consisting of red / green / blue (RGB) components. For each of these three colors, 256 (0-255) values are available, which can be entered here statically. The color can be selected from a selection list or via the color selection dialog, which can be opened via the button



. In addition, the level of transparency can be set for each color (0: fully transparent, 255: fully opaque).

Color areas

Durable color areas	If this option is enabled, the color areas are permanently visible. The effects of this option are visible only in online mode.
[<n>]	Click on the  Create new button to generate a new color area. For each color area an area is created that covers the corresponding settings. [<n>]: The number indexes the area. Clicking 'Delete' deletes the corresponding color area and its settings.

Area [<n>]

Begin of area	Start of the color range. It must lie within the defined <u>scale</u> [▶ 539].
End of area	End of the color range. It must lie within the defined <u>scale</u> [▶ 539].
Color	Color of the bar area



Transparency is not supported under Windows CE.

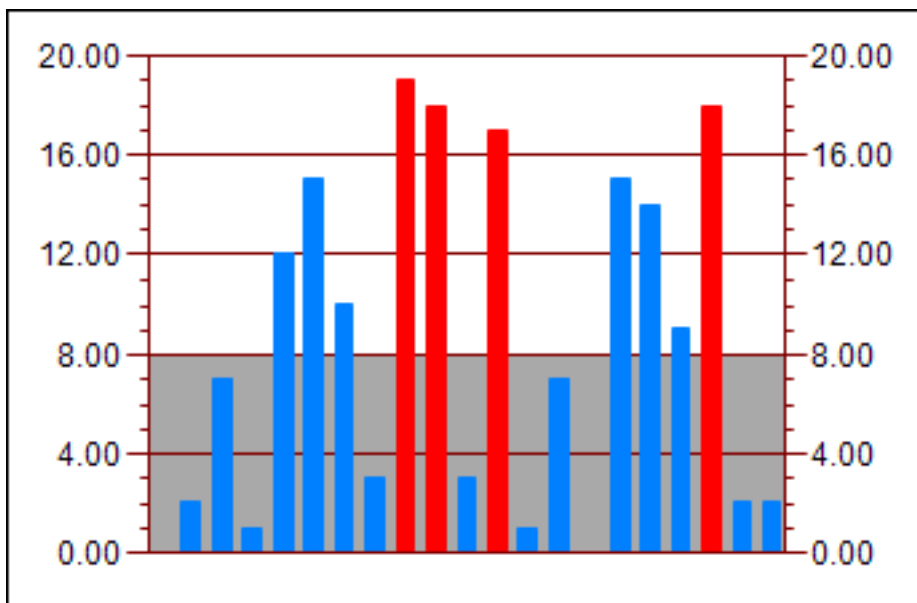
Access rights

This setting relates to the access rights for the individual element. Click to open the Access rights dialog [▶ 420]. The setting is only available if a user management [▶ 393] was added to the PLC project. The following status messages are available:

Not set. All rights.	The default message is set, if the element is shown as available for all groups.
Rights are issued: Limited rights.	The message is set, if the element is shown with limited behavior for at least one group.

15.8.5.6 Histogram

This element can be used to add a histogram to a visualization page for displaying the values of an array. The minimum and maximum display values can be specified here. For certain value ranges special colors [▶ 576] can be defined. An example of the configuration can be found in the section "Configuration of a histogram [▶ 577]".



Properties editor

The properties of a visualization element - except alignment and order [▶ 377] - can all be configured in the properties editor [▶ 385]. By default, this editor opens next to the visualization editor, or it can be opened explicitly via the "Properties" command (which can be found in the View menu as standard).

A property can be modified by editing the field "Value". To this end, an input field, a selection list, a dialog or checkbox that can be activated is provided in this field, depending on the element type. The value field opens

- after a double-click,
- after a single click in a selected field,
- via the space bar, if the field was already selected.

If a variable is assigned,

- simply enter its name.
- Use the button to open the input assistant for selecting a variable. The Variables category lists all variables that have already been defined in the project.

Working in the list of properties can be made easier with the aid of default, sorting and filter functions.

Element properties

All element properties and their descriptions are listed below.

Element name	The element name can be changed. Standard name is "GenElemInst_x". "x" stands for a sequential number.
Element type	The element type is entered here. For three element groups it is possible to switch between the corresponding elements by changing the element type: <ul style="list-style-type: none"> • <u>rectangle, rounded rectangle and ellipse</u> [▶ 475] • <u>Polygon, polyline or Bézier curve</u> [▶ 490] • <u>dip switch, power switch, push switch, push switch with LED, rocker switch</u> [▶ 532]

Auxiliary setting

Data array	Variable of type array, whose data is to be visualized.
-------------------	---

Subrange of array


Use subrange	If this option is enabled, only the subrange of the assigned data array is used
Start index	Indexes the first data set of the assigned data array.
End index	Indexes the last data set of the assigned data array.
Display type	Here you can select the display type of the data in the histogram: <ul style="list-style-type: none"> • Bars • Lines • Curve
Line width	Thickness of the curve representing the data in the histogram. This setting is only available if the display type "Curve" was selected.
Show horizontal lines	If this option is enabled, horizontal lines are drawn at the main scales.
Relative bar width	Relative width of the bar or the line in pixels

Position



Here you can define the position (X/Y coordinates) and size (width and height) of the element in pixels. The origin is in the top left corner of the window. The positive x-axis is on the right, the positive y-axis runs downwards. If the values edited, the displayed element is simultaneously modified in the [visualization editor](#) [▶ 376].

X	Horizontal position in pixels – X=0 is the left edge of the window.
Y	Vertical position in pixels – Y=0 is the upper edge of the window.
Width	Width of the element in pixels
Height	Height of the element in pixels

Scale

Scale start	Lower scale limit value
Scale end	Upper scale limit value
Main scale	Distance between two lines of the coarse scale
Sub scale	Distance between two lines on the fine scale. The value can be set to 0 if a further subdivision of the coarse scale is not desired.
Scale color	The color can be set via the selection list or via the  button from the standard color selection dialog.
Base line	Numeric value by which the base line of the histogram is to be moved upwards.

Labelling


Unit	The entered text is used as element label. It is shown below the center of the scale and can be used to specify the unit of the scale, for example.
Font	Here you can set the font for the unit and the scale: <ul style="list-style-type: none"> • Standard • Heading • Large • Title • Note Click the  button to open a dialog for user-defined font property settings.
Scale format (C-syntax)	Use the C syntax to specify the formatting of the scale label. For example, entering the string "%3.2f s" in this field results in scale labels with 3 digits, 2 of which are decimal places, followed by the letter "s".
Max. text width of labels	Value specifying the maximum width of the scale label. This value is generally set correctly automatically. Only use this setting option if the automatic adjustment does not lead to the required result.
Text height of labels	Value specifying the height of the scale label. This value is generally set correctly automatically. Only use this setting option if the automatic adjustment does not lead to the required result.
Font color	The color for the label can be set via the selection list or via the  button from the standard color selection dialog.

Colors

A color is defined based on a hexadecimal number consisting of red / green / blue (RGB) components. For each of these three colors, 256 (0-255) values are available, which can be entered here statically. The color can be selected from a selection list or via the color selection dialog, which can be opened via the button



. In addition, the level of transparency can be set for each color (0: fully transparent, 255: fully opaque).

Graph color	Color for the graph
Alarm color <ul style="list-style-type: none"> • Alarm value • Alarm condition • Alarm color 	<ul style="list-style-type: none"> • Alarm value Here you can specify the threshold value for the alarm. <ul style="list-style-type: none"> • Alarm condition The alarm is set if the current value of the array element meets the alarm condition. If "Less" is set, the alarm is triggered if the value is lower than the threshold value. If "More" is set, the alarm is triggered, if the value is greater than the threshold value. <ul style="list-style-type: none"> • Alarm color Here you can select a color for displaying the individual bar in the event of an alarm.
Use color areas	If this option is enabled, the color areas are displayed as defined under color areas.
Color areas <ul style="list-style-type: none"> • [<n>] 	Click on the  button to generate a new color area. For each color area an area is created that covers the corresponding settings. [<n>]: The number indexes the area. Clicking "Delete" deletes the corresponding color area and its settings.

Area [<n>]

Begin of area	Start of the color range. It must lie within the defined scale [► 539].
End of area	End of the color range. It must lie within the defined scale [► 539].
Color	Color of the bar area



Transparency is not supported under Windows CE.

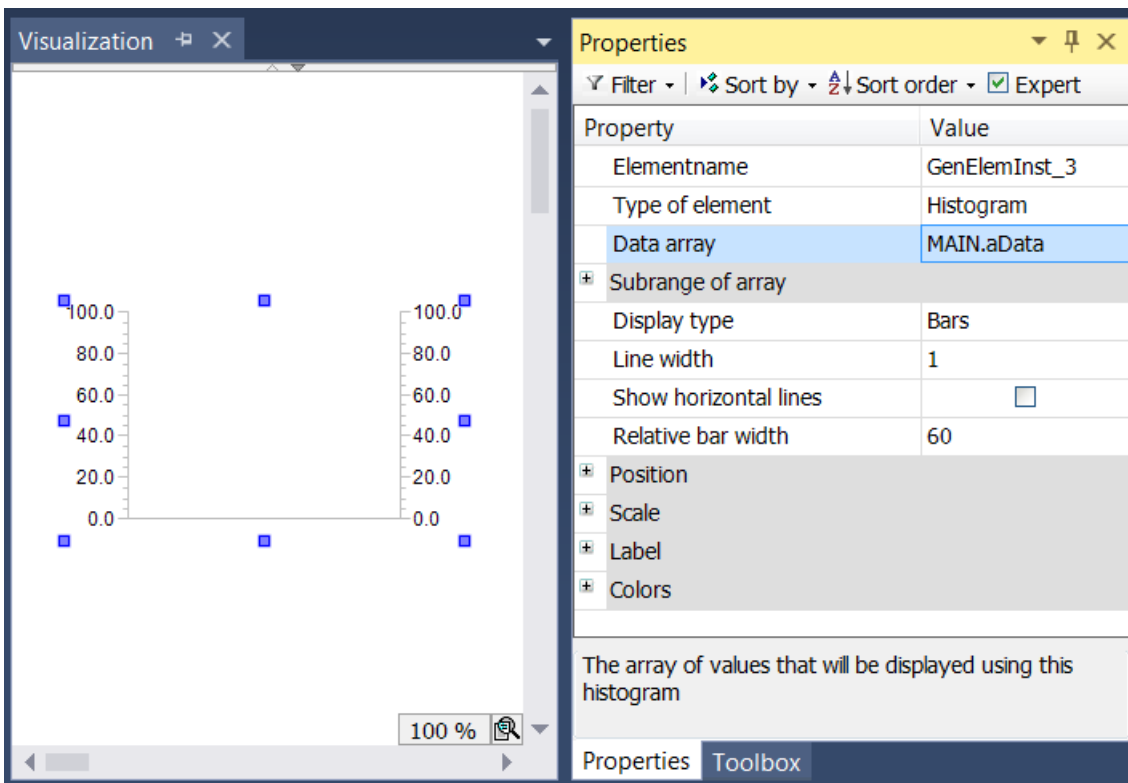
Access rights

This setting relates to the access rights for the individual element. Click to open the [Access rights dialog](#) [▶ 420]. The setting is only available if a [user management](#) [▶ 393] was added to the PLC project. The following status messages are available:

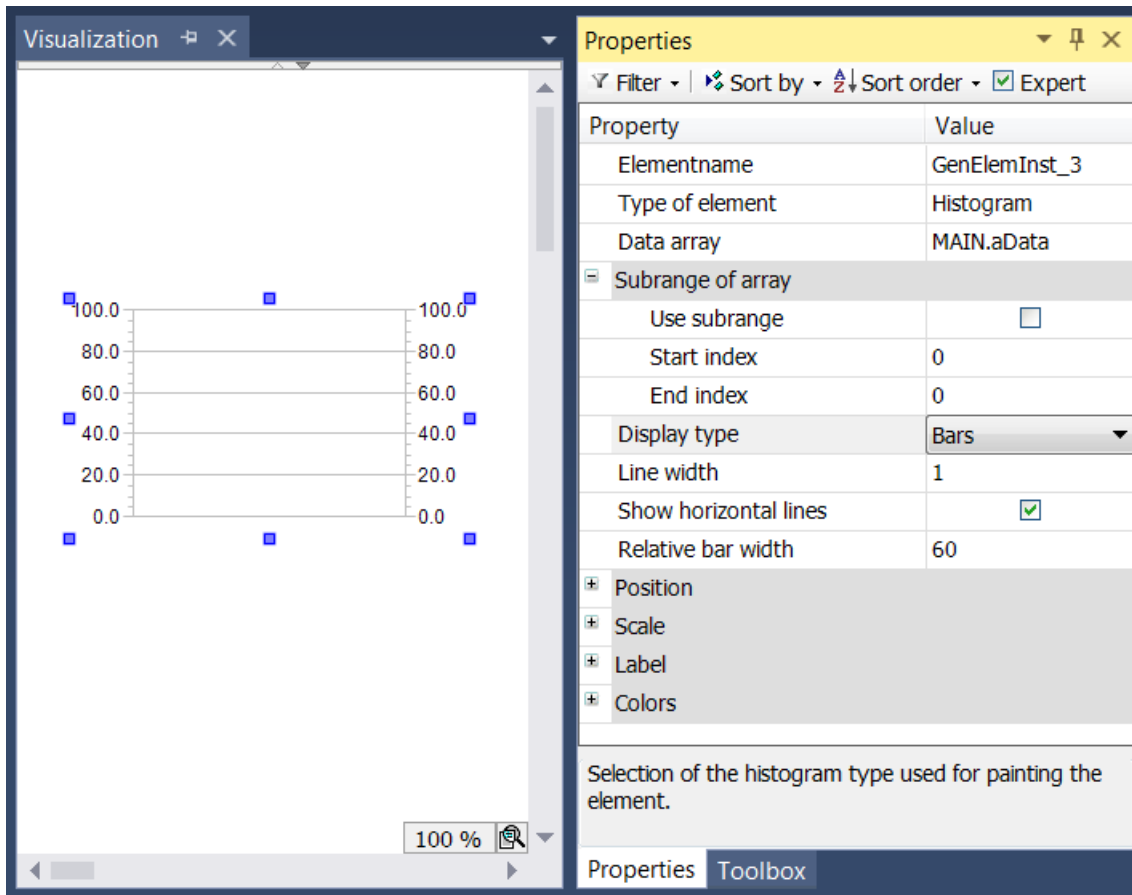
Not set. All rights.	The default message is set, if the element is shown as available for all groups.
Rights are issued: Limited rights.	The message is set, if the element is shown with limited behavior for at least one group.

15.8.5.6.1 Configuration of a histogram

The following section explains the configuration of a histogram, based on an example.



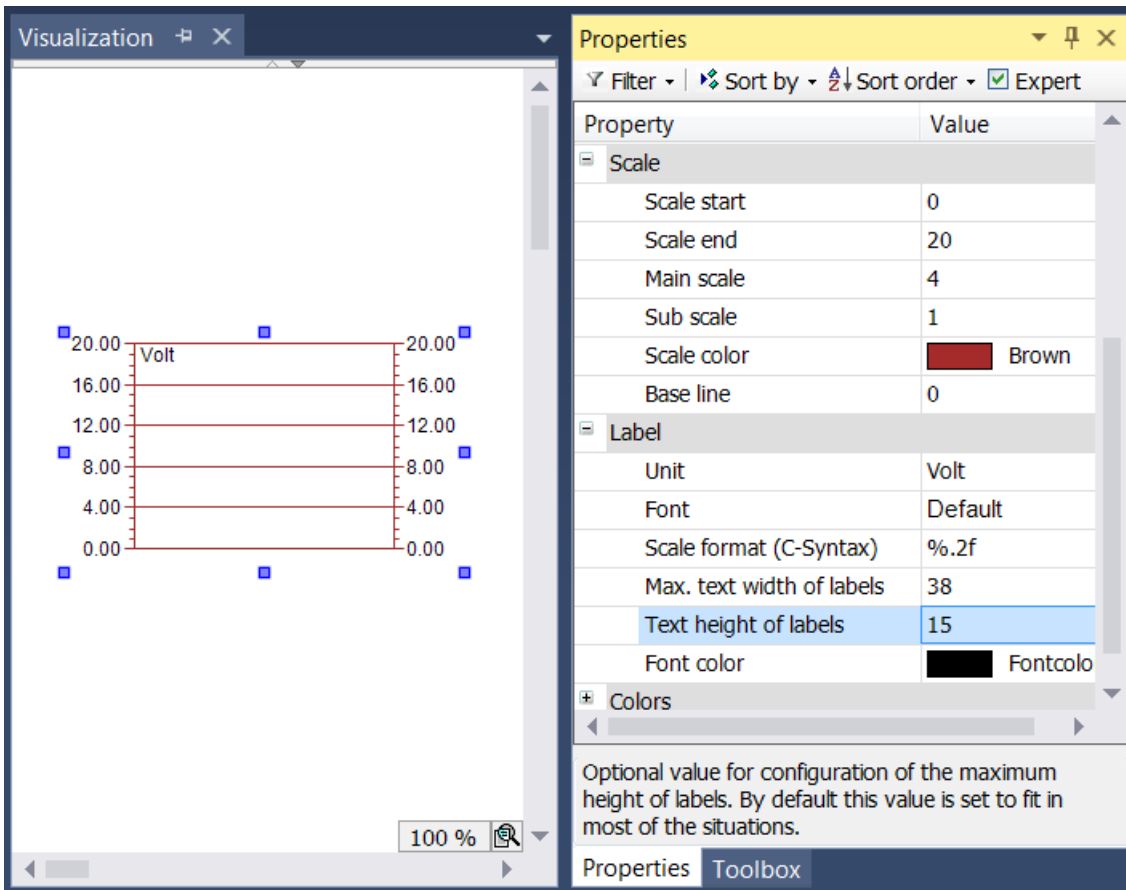
The assigned data array – in the example an array with the name "aData" – must be specified in the element properties of the histogram element. The button is available after clicking in the input field of the "Data array" property. It can be used to search for the variable in the project. Be sure to specify the input variable with its full path in the device tree.



The option "Use subrange" is used if not all elements of the arrays are to be displayed, but only the elements from "Start index" to "End index". In addition, the display type of the array data can be shown in three different forms:

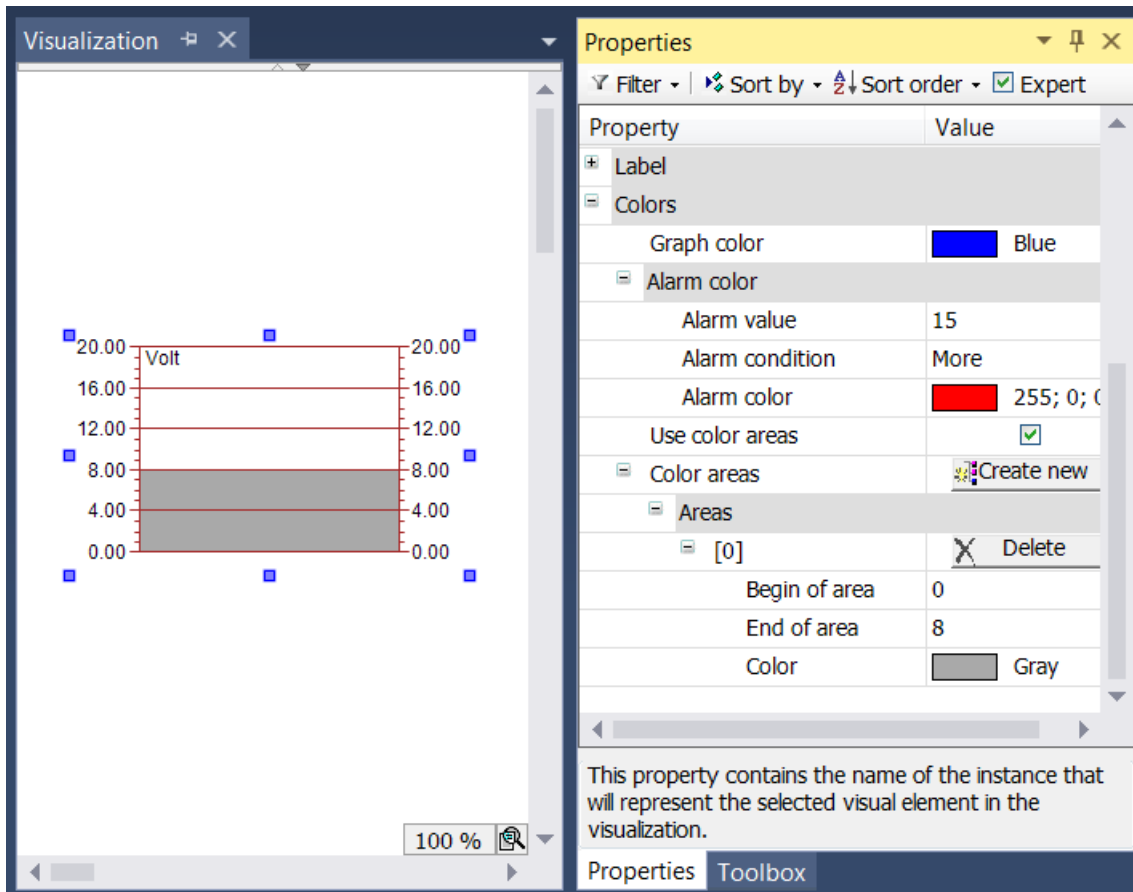
Bars	
Lines	
Curve	

In this histogram example, horizontal lines are drawn at the main scales.

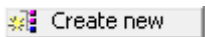



In the scale properties the numeric range of the scale and the main and sub scales can be defined. The scale range is limited by the values specified in "Scale start" and "Scale end". The value of "Scale start" must therefore be lower than that of "Scale end". The values for Main scale and Sub scale can be set to 0, in order to disable the display of the scale lines: if the value for the main scale is set to 0, no sub scale lines are drawn, irrespective of the value set for the sub scale. If the value for the sub scale is set to 0, so, only the scale lines for the main scale are drawn. To change the scale color, click on the field for the scale color. A dialog opens, in which the color can be selected.

The scale display can be specified in the Label section. The entry in the input field Unit is intended for specification of the scale unit. It is displayed at the top left of the histogram. After selecting a suitable font and font color, the label format can be adjusted. The formatting of the numerical values must be specified according to the syntax the C programming language. Use "%d" for integer values and "%.Xf" for floating-point numbers, whereby "X" should be replaced by the required number of decimal places.



Finally, the element color can be specified in the 'Alarm color' section. First, select the color for the histogram. Within the subsection "Alarm color" an alarm color can be defined in which the individual bar is displayed, if the current value of the array element meets the alarm condition. This condition results from the definition of the alarm value, which must either not be exceeded (if the alarm condition is set to "More"), or the value must not fall below a minimum value (if the alarm condition is set to "Less").

Subareas of the scale can be assigned a certain color by creating a color area with the  button. The color areas are numbered in ascending order. Each color area is assigned its own input fields within the element properties.

The limits of the subarea can be specified under "Begin of area" and "End of area". The color can be selected via a pull-down menu. Once a color area has been created, it can be deleted by clicking on the corresponding  button.

15.8.6 Special controls

15.8.6.1 Flower waiting icon

This animated element can be used to indicate that the system is busy or of waiting for data.




Properties editor

The properties of a visualization element - except [alignment and order \[▶ 377\]](#) - can all be configured in the [properties editor \[▶ 385\]](#). By default, this editor opens next to the visualization editor, or it can be opened explicitly via the "Properties" command (which can be found in the View menu as standard).

A property can be modified by editing the field "Value". To this end, an input field, a selection list, a dialog or checkbox that can be activated is provided in this field, depending on the element type. The value field opens

- after a double-click,
- after a single click in a selected field,
- via the space bar, if the field was already selected.

If a variable is assigned,

- simply enter its name.
- Use the  button to open the input assistant for selecting a variable. The Variables category lists all variables that have already been defined in the project.

Working in the list of properties can be made easier with the aid of default, sorting and filter functions.

Element properties

All element properties and their descriptions are listed below.

Element name	The element name can be changed. Standard name is "GenElemInst_x". "x" stands for a sequential number.
Element type	The element type is entered here. For three element groups it is possible to switch between the corresponding elements by changing the element type: <ul style="list-style-type: none"> • rectangle, rounded rectangle and ellipse [▶ 475] • Polygon, polyline or Bézier curve [▶ 490] • dip switch, power switch, push switch, push switch with LED, rocker switch [▶ 532]

Position

Here you can define the position (X/Y coordinates) and size (width and height) of the element in pixels. The origin is in the top left corner of the window. The positive x-axis is on the right, the positive y-axis runs downwards. If the values edited, the displayed element is simultaneously modified in the [visualization editor \[▶ 376\]](#).

X	Horizontal position in pixels – X=0 is the left edge of the window.
Y	Vertical position in pixels – Y=0 is the upper edge of the window.
Width	Width of the element in pixels
Height	Height of the element in pixels

Colors

A color is defined based on a hexadecimal number consisting of red / green / blue (RGB) components. For each of these three colors, 256 (0-255) values are available, which can be entered here statically. The color can be selected from a selection list or via the color selection dialog, which can be opened via the button



. In addition, the level of transparency can be set for each color (0: fully transparent, 255: fully opaque).

Frame color	Select a frame color for the element.
Fill color	Select a fill color for the element.



Transparency is not supported under Windows CE.

Appearance

The settings under Appearance are static definitions for the frame and the element filling.

Line width	Defines the width of the frame line in pixels. 0 codes the same as 1 and sets the line width to 1 pixel. If no frame is required, set the line type to invisible.
Fill type	Defines the fill type for the fill color: <ul style="list-style-type: none"> • Filled: The fill color is visible • Invisible: The fill color is invisible
Line type	Defines one of the following line types for the outline: <ul style="list-style-type: none"> • solid • dashed • Points • dotdash • dash dot dot • Invisible: outline is invisible.
Icon color	Here you can select a color for displaying the flower icon.
Icon line width	Here you can specify a number to indicate the width of the icon lines. 0 codes the same as 1 and sets the line width to 1 pixel.

State variables

These are dynamic definitions of the availability of the element in online mode.

Invisibility	Boolean variable. If this returns TRUE, the element is invisible in online mode.
--------------	--

Access rights

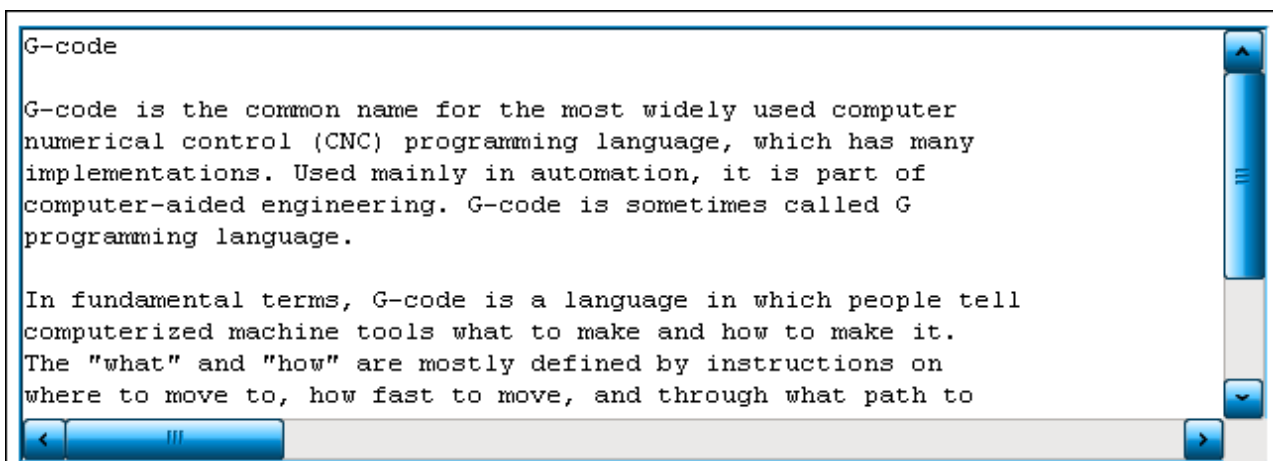
This setting relates to the access rights for the individual element. Click to open the [Access rights dialog \[▶ 420\]](#). The setting is only available if a [user management \[▶ 393\]](#) was added to the PLC project. The following status messages are available:

Not set. All rights.	The default message is set, if the element is shown as available for all groups.
Rights are issued: Limited rights.	The message is set, if the element is shown with limited behavior for at least one group.

15.8.6.2 Text Editor

The visualization element Text editor is used to display and edit the content of text files in ASCII or Unicode, which are located on the controller.

- [User inputs for navigation \[▶ 585\]](#)
- [Configuration example \[▶ 586\]](#)





The Text editor element can only be used in combination with the [PLC HMI \[▶ 603\]](#) and/or the [PLC HMI Web \[▶ 608\]](#).


Properties editor

The properties of a visualization element - except [alignment and order \[▶ 377\]](#) - can all be configured in the [properties editor \[▶ 385\]](#). By default, this editor opens next to the visualization editor, or it can be opened explicitly via the "Properties" command (which can be found in the View menu as standard).

A property can be modified by editing the field "Value". To this end, an input field, a selection list, a dialog or checkbox that can be activated is provided in this field, depending on the element type. The value field opens

- after a double-click,
- after a single click in a selected field,
- via the space bar, if the field was already selected.

If a variable is assigned,

- simply enter its name.
- Use the  button to open the input assistant for selecting a variable. The Variables category lists all variables that have already been defined in the project.

Working in the list of properties can be made easier with the aid of default, sorting and filter functions.

Element properties

All element properties and their descriptions are listed below.

Element name	The element name can be changed. Standard name is "GenElemInst_x". "x" stands for a sequential number.
Element type	The element type is entered here. For three element groups it is possible to switch between the corresponding elements by changing the element type: <ul style="list-style-type: none"> • rectangle, rounded rectangle and ellipse [▶ 475] • Polygon, polyline or Bézier curve [▶ 490] • dip switch, power switch, push switch, push switch with LED, rocker switch [▶ 532]

Position

Here you can define the position (X/Y coordinates) and size (width and height) of the element in pixels. The origin is in the top left corner of the window. The positive x-axis is on the right, the positive y-axis runs downwards. If the values edited, the displayed element is simultaneously modified in the [visualization editor \[▶ 376\]](#).

X	Horizontal position in pixels – X=0 is the left edge of the window.
Y	Vertical position in pixels – Y=0 is the upper edge of the window.
Width	Width of the element in pixels
Height	Height of the element in pixels

Font

Name	Name of a font installed on the system. The text is output in the text editor with this font. A non-proportional font such as Courier New must be used.
Size	Font size Example: 12

Control variable**File**

File name	STRING variable containing the file name, and, if necessary, the path
Open	Boolean variable for controlling the opening of the file specified in "File name". If the variable is set to TRUE, the file opens.
Close	Boolean variable for controlling the closing of the file. The file closes if the variable is set to TRUE.
Save	Boolean variable for controlling the saving of the file. The file is saved if the variable is set to TRUE.
New	Boolean variable for controlling the generation of a new file. If the variable is set to TRUE, a new file is created with the name defined in "File name".

Edit

Search for	STRING variable containing the search term
Find	Boolean variable that controls the search for the search term. The search starts if the variable is set to TRUE.
Find next occurrence	Boolean variable for controlling the search for the next occurrence of the search term. If the variable is set to TRUE, the search starts at the current position.

Caret position

Row	Integer variable that contains the current row number, as long as "Trigger setting" is FALSE.
Column	Integer variable that contains the current column number, as long as "Trigger setting" is FALSE.
Position	Integer variable that contains the current caret position in consecutive numbering, as long as "Trigger setting" is FALSE.
Trigger setting	Boolean variable that controls the caret position. If the variable is FALSE, the variables in "Row", "Column" and "Position" contain the current position. If the variable is TRUE, the caret is set to the position specified in "Row" and 'Column'.

Selection

Start position	Integer variable containing the start position of the text selection in consecutive numbering.
End position	Integer variable containing the end position of the text selection in consecutive numbering.
Start line number	Integer variable containing the line number at the start of the text selection.
Start column index	Integer variable containing the column index at the start of the text selection.
End line number	Integer variable containing the line number at the end of the text selection.
End column index	Integer variable containing the column index at the end of the text selection.
Line to be selected	Integer variable determining a line to be selected
Trigger selection	Boolean variable for controlling the text selection. If the variable is set to TRUE, the text selection is set as defined in "Row to be selected".

Error handling

Variable for error number	Integer variable that contains the error number in case of error. The numbers are declared in GVL_ErrorCodes, part of the library "VisuElemTextEditor". To obtain the error text associated with the error number, call the function VisuFctTextEditorGetErrorText(), which is contained in the library.
---------------------------	--

Variable changed for content	Boolean variable. The content of the text editor has changed if the variable value is TRUE.
Variable for access mode	Boolean variable. If the variable value is TRUE, the file has read access only. If the variable value is FALSE, the file has read and write access.
Maximum line length	Integer variable for specifying the maximum length of a line
Edit mode	<p>Here can select the edit mode:</p> <ul style="list-style-type: none"> • Read only <p>The file can only be read. In this mode the editor is shown with a light grey background, if the controller is running.</p> <ul style="list-style-type: none"> • Read/write <p>The file can be read and written.</p>

New files

Character encoding	<p>The character encoding when a new file is created can be specified here:</p> <ul style="list-style-type: none"> • ASCII • Unicode (Little endian) • Unicode (Big endian)
Line end character	<p>If a new file is created, the line end character must be defined here:</p> <ul style="list-style-type: none"> • CR/LF: standard in Windows systems • LF: standard in UNIX systems <p>If an existing file is opened, the line end character of the file to be opened is identified and used automatically.</p>

Access rights

This setting relates to the access rights for the individual element. Click to open the [Access rights dialog \[► 420\]](#). The setting is only available if a [user management \[► 393\]](#) was added to the PLC project. The following status messages are available:

Not set. All rights.	The default message is set, if the element is shown as available for all groups.
Rights are issued: Limited rights.	The message is set, if the element is shown with limited behavior for at least one group.

15.8.6.2.1 User inputs for navigation in online mode

The text editor enables generally known user inputs for navigating in the text. These are available only in online mode.

[Left arrow]	The caret is moved one character to the left. If the current caret position is at the start of a line, it is moved to the end the previous line, if present.
[Right arrow]	The caret is moved one character to the right. If the current caret position is at the end of a line, it is moved to the start of the next line, if present.
[Up arrow]	The caret is moved to the previous line.
[Down arrow]	The caret is moved to the next line.
[Home]	The caret is moved to the start of the current line.
[End]	The caret is moved to the end of the current line.
[PgUp]	The previous page is displayed.
[PgDn]	The next page is displayed.
[Shift] + [Left arrow]	Text selection
[Shift] + [Right arrow]	
[Del] when text is selected	The selected text is deleted. Is nothing is selected, the character to the right of the caret position is deleted.
[Enter]	A new line is created, and the caret is set to the start of the new line.
[Ctrl] + [Tab]	A tab character is inserted. Currently it is represented as a single blank.
[Tab]	The focus is moved to the next visualization.

15.8.6.2.2 Configuration of a text editor

To access the control variables supported by the `text editor` [► 582], it is necessary to add an additional visualization element to the visualization and link it with the text editor via control variables.



The procedure is described below, using the example of the loading functionality of the text editor:

1. Declaration of the control variable for the element "Load" in IEC code in MAIN, for example:

```
PROGRAMM MAIN
VAR
  bOpen : BOOL;
END_VAR
```

2. Declaration of the variable for the file name of the text editor in IEC code in MAIN, for example:

```
PROGRAMM MAIN
VAR
  bOpen : BOOL;
  sFileName : STRING;
END_VAR
```

3. Adding a rectangle element to visualization and its configuration:

- Texts → Text : Load

- Input configuration → Keys → Variable : MAIN.bOpen

4. Adding a text editor element to the visualization and its configuration:

- Control variable → File → File name : MAIN.sFileName
- Control variable → File → Open : MAIN.bOpen

All control variables that are present in the properties of the text editor can be linked on this way.

Error messages

To output the error text for the error number that is available in Control variable → Error handling → Variable for error number, program the function call `VisuFctTextEditorGetErrorText()` in IEC code:

1. Declaration of the required variable:

```
PROGRAMM MAIN
VAR
...
nErrorCode : USINT;
sErrorMessage : STRING(255);
END_VAR
```

2. Implementation of the function call:

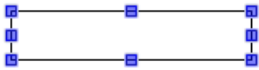
```
sErrorMessage := VisuFctTextEditorGetErrorText(nErrorCode);
```

3. Output of the error message in the visualization, for example in a rectangle element with the following properties:

- Texts → Text : %s
- Text variables → Text variable : MAIN.sErrorMessage

15.8.6.3 ActiveX element

The ActiveX element enables integration of an existing ActiveX component.



ActiveX control can only be used in combination with the [integrated visualization \[▶ 602\]](#) and not in combination with the [PLC HMI \[▶ 603\]](#) or [PLC HMI Web \[▶ 608\]](#).


Properties editor

The properties of a visualization element - except [alignment and order \[▶ 377\]](#) - can all be configured in the [properties editor \[▶ 385\]](#). By default, this editor opens next to the visualization editor, or it can be opened explicitly via the "Properties" command (which can be found in the View menu as standard).

A property can be modified by editing the field "Value". To this end, an input field, a selection list, a dialog or checkbox that can be activated is provided in this field, depending on the element type. The value field opens

- after a double-click,
- after a single click in a selected field,
- via the space bar, if the field was already selected.

If a variable is assigned,

- simply enter its name.
- Use the  button to open the input assistant for selecting a variable. The Variables category lists all variables that have already been defined in the project.


Working in the list of properties can be made easier with the aid of default, sorting and filter functions.

Element properties

All element properties and their descriptions are listed below.

Element name	The element name can be changed. Standard name is "GenElemInst_x". "x" stands for a sequential number.
Element type	The element type is entered here. For three element groups it is possible to switch between the corresponding elements by changing the element type: <ul style="list-style-type: none"> • rectangle, rounded rectangle and ellipse [▶ 475] • Polygon, polyline or Bézier curve [▶ 490] • dip switch, power switch, push switch, push switch with LED, rocker switch [▶ 532]

Auxiliary setting

Element	An installed ActiveX component can be assigned here. Use the  button to select a component via the input wizard. Enter a variable of type string, which describes the name or the ID of the component.
----------------	---

Position

Here you can define the position (X/Y coordinates) and size (width and height) of the element in pixels. The origin is in the top left corner of the window. The positive x-axis is on the right, the positive y-axis runs downwards. If the values edited, the displayed element is simultaneously modified in the [visualization editor \[▶ 376\]](#).

X	Horizontal position in pixels – X=0 is the left edge of the window.
Y	Vertical position in pixels – Y=0 is the upper edge of the window.
Width	Width of the element in pixels
Height	Height of the element in pixels

Absolute movement

The element can be moved by changing the x- and y-position (pixels) of the top left corner of the element via an integer variable. Absolute coordinate values are used here.

Motion	X: The integer variable entered here defines the current x-position of the top left corner of the element (in pixels). It can be used to move the element in x-direction. (A positive value moves the element from left to right).
• X	
• Y	Y: The integer variable entered here defines the current y-position of the top left corner of the element (in pixels). It can be used to move the element in y-direction. (A positive value moves the element from top to bottom).



State variables

These are dynamic definitions of the availability of the element in online mode.

Invisibility	Boolean variable. If this returns TRUE, the element is invisible in online mode.
--------------	--

Initial calls



Method calls that are executed during the initialization can be defined under this property. They are only executed in the first cycle.

Use the  button to create new method calls and associated parameters. The  button can be used to delete the corresponding method call with all settings, such as parameters and result parameters.

[<Number>]	Number that indexes a method call
Method	Name of the method to be called
Parameter	Parameter variables
Result parameter	Variable that is to contain the result. A result parameter is basically available to every method.

Cyclic calls



Method calls that are executed in each cycle can be defined under this property. They are called during the update interval of the visualization.

Use the  button to create new method calls and associated parameters. The  button can be used to delete the corresponding method call with all settings, such as parameters and result parameters.

[<Number>]	Number that indexes a method call
Method	Name of the method to be called
Parameter	Parameter variables
Result parameter	Variable that is to contain the result. A result parameter is basically available to every method.

Conditional calls

Method calls that are only executed under certain conditions can be defined under this property. They are called during the update interval of the visualization. In contrast to cyclic calls, a call condition is defined under the property "Call condition". They are only executed with a rising edge of the call condition.

Use the  button to create new method calls and associated parameters. The  button can be used to delete the corresponding method call with all settings, such as parameters and result parameters.

[<Number>]	Number that indexes a method call
Method	Name of the method to be called
Call condition	Boolean variable. The assigned method is called once exclusively on a rising edge of this variable.
Parameter	Parameter variables
Result parameter	Variable that is to contain the result. A result parameter is basically available to every method.

Access rights

This setting relates to the access rights for the individual element. Click to open the [Access rights dialog](#) [[▶ 420](#)]. The setting is only available if a [user management](#) [[▶ 393](#)] was added to the PLC project. The following status messages are available:

Not set. All rights.	The default message is set, if the element is shown as available for all groups.
Rights are issued: Limited rights.	The message is set, if the element is shown with limited behavior for at least one group.

15.8.6.4 Webbrowser

The Webbrowser element shows content that is specified the via a URL address. In addition to HTML pages, it can also load videos and PDF documents. An example of the configuration can be found in the section "[Configuration](#) [[▶ 591](#)]".




Properties editor

The properties of a visualization element - except alignment and order [[▶ 377](#)] - can all be configured in the properties editor [[▶ 385](#)]. By default, this editor opens next to the visualization editor, or it can be opened explicitly via the "Properties" command (which can be found in the View menu as standard).

A property can be modified by editing the field "Value". To this end, an input field, a selection list, a dialog or checkbox that can be activated is provided in this field, depending on the element type. The value field opens

- after a double-click,
- after a single click in a selected field,
- via the space bar, if the field was already selected.

If a variable is assigned,

- simply enter its name.
- Use the  button to open the input assistant for selecting a variable. The Variables category lists all variables that have already been defined in the project.

Working in the list of properties can be made easier with the aid of default, sorting and filter functions.

Element properties

All element properties and their descriptions are listed below.

Element name	The element name can be changed. Standard name is "GenElemInst_x". "x" stands for a sequential number.
Element type	The element type is entered here. For three element groups it is possible to switch between the corresponding elements by changing the element type: <ul style="list-style-type: none"> • <u>rectangle, rounded rectangle and ellipse</u> [▶ 475] • <u>Polygon, polyline or Bézier curve</u> [▶ 490] • <u>dip switch, power switch, push switch, push switch with LED, rocker switch</u> [▶ 532]

Position

Here you can define the position (X/Y coordinates) and size (width and height) of the element in pixels. The origin is in the top left corner of the window. The positive x-axis is on the right, the positive y-axis runs downwards. If the values edited, the displayed element is simultaneously modified in the visualization editor [[▶ 376](#)].

X	Horizontal position in pixels – X=0 is the left edge of the window.
Y	Vertical position in pixels – Y=0 is the upper edge of the window.
Width	Width of the element in pixels
Height	Height of the element in pixels

Absolute movement

The element can be moved by changing the x- and y-position (pixels) of the top left corner of the element via an integer variable. Absolute coordinate values are used here.

Motion	X: The integer variable entered here defines the current x-position of the top left corner of the element (in pixels). It can be used to move the element in x-direction. (A positive value moves the element from left to right).
<ul style="list-style-type: none"> • X • Y 	Y: The integer variable entered here defines the current y-position of the top left corner of the element (in pixels). It can be used to move the element in y-direction. (A positive value moves the element from top to bottom).

State variables

These are dynamic definitions of the availability of the element in online mode.

Invisibility	Boolean variable. If this returns TRUE, the element is invisible in online mode.
--------------	--

Control variable

URL	Variable of type string, in which the URL address of the website to be opened is stored Example: sUrlAddress : STRING := 'http://www.beckhoff.com/';
Show	Boolean variable. If the variable contains a rising edge, the visualization calls the web page specified in URL and displays its content in the element's frame.
Back	Boolean variable. If the variable contains a rising edge, the visualization displays the content of the previously displayed page.
Forward	Boolean variable. If the variable contains a rising edge, the visualization displays the content from before the back navigation.

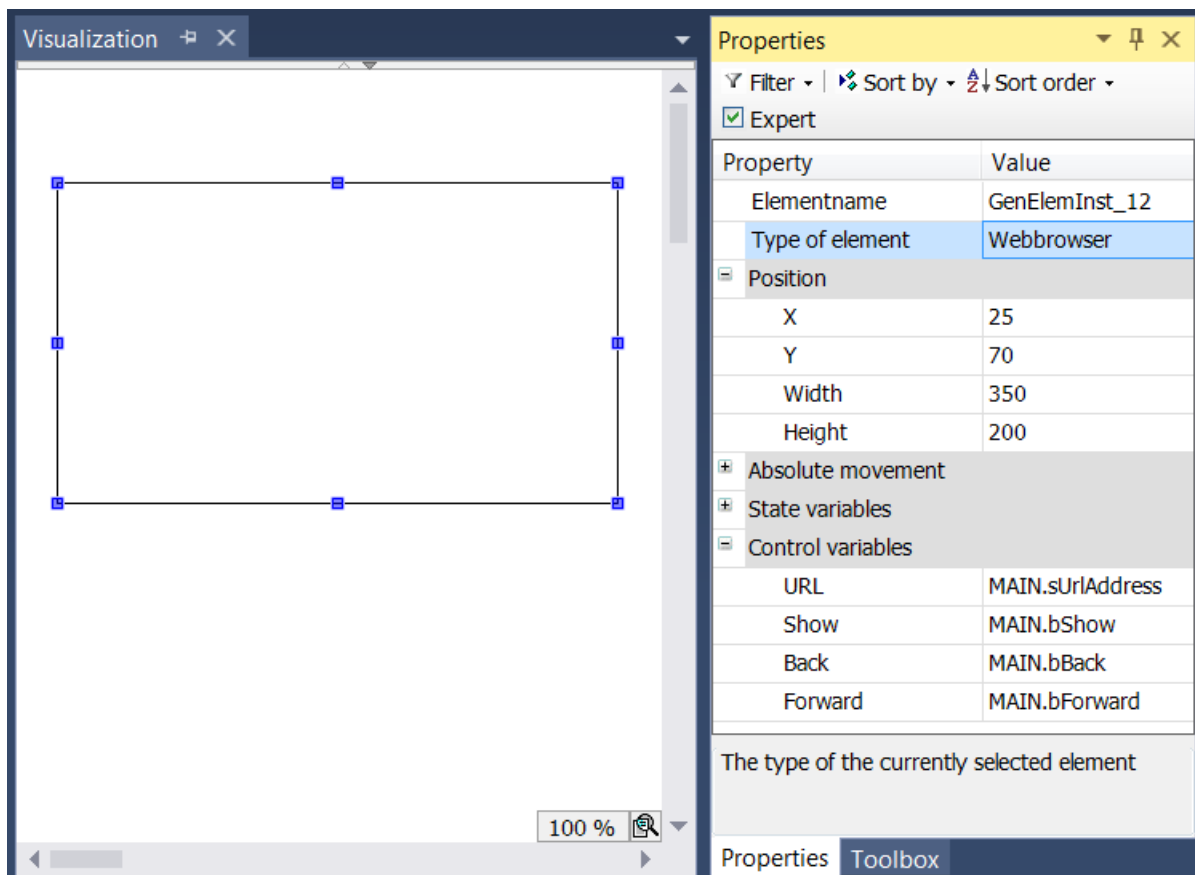
Access rights

This setting relates to the access rights for the individual element. Click to open the [Access rights dialog](#) [► 420]. The setting is only available if a [user management](#) [► 393] was added to the PLC project. The following status messages are available:

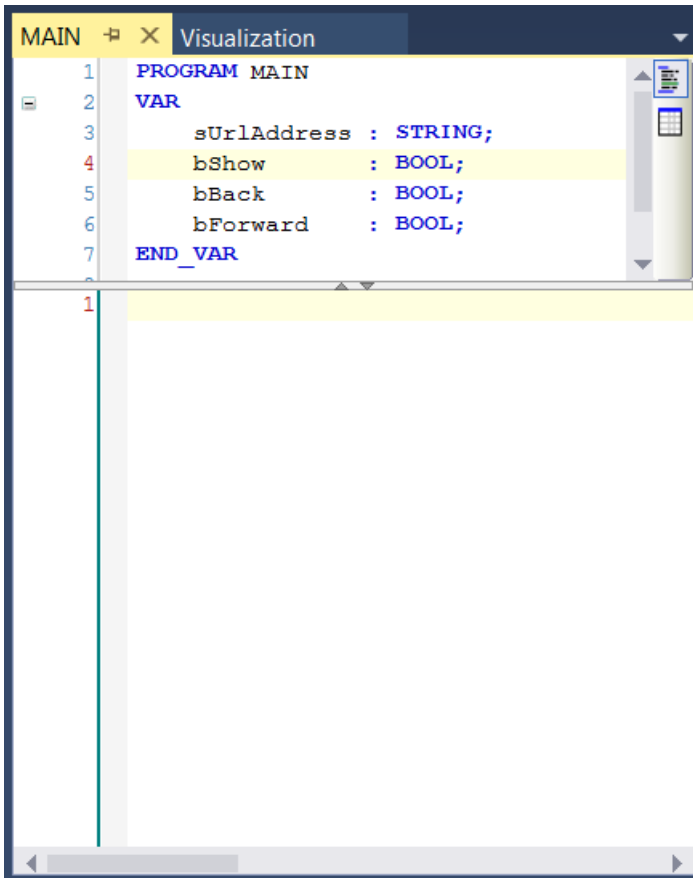
Not set. All rights.	The default message is set, if the element is shown as available for all groups.
Rights are issued: Limited rights.	The message is set, if the element is shown with limited behavior for at least one group.

15.8.6.4.1 Configuration

The following section explains the configuration of a Webbrowser element, based on an example.



Once the Webbrowser element has been dragged from the toolbox to the [visualization page \[▶ 402\]](#), the position and size of the element can be modified via the property "Position". In addition, variables for operating the browser can be specified in the "[control variables \[▶ 591\]](#)". In this example the variables "sUrlAddress", "bShow", "bBack" and "bForward" that have been declared in the "MAIN" program are entered:



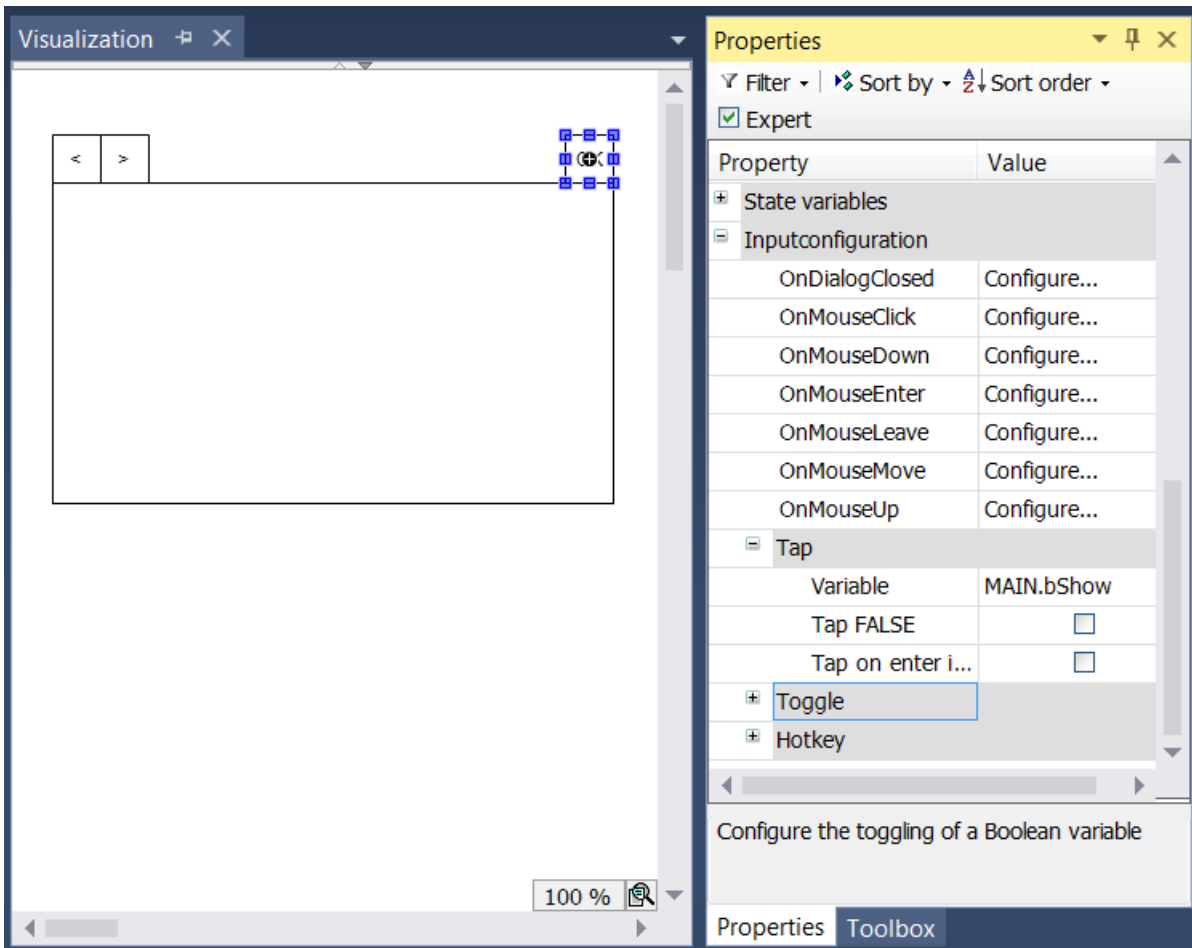
```
1 PROGRAM MAIN
2 VAR
3     sUrlAddress : STRING;
4     bShow       : BOOL;
5     bBack       : BOOL;
6     bForward    : BOOL;
7 END_VAR
```

The screenshot shows a window titled 'MAIN Visualization'. The top part is a code editor with the following content:

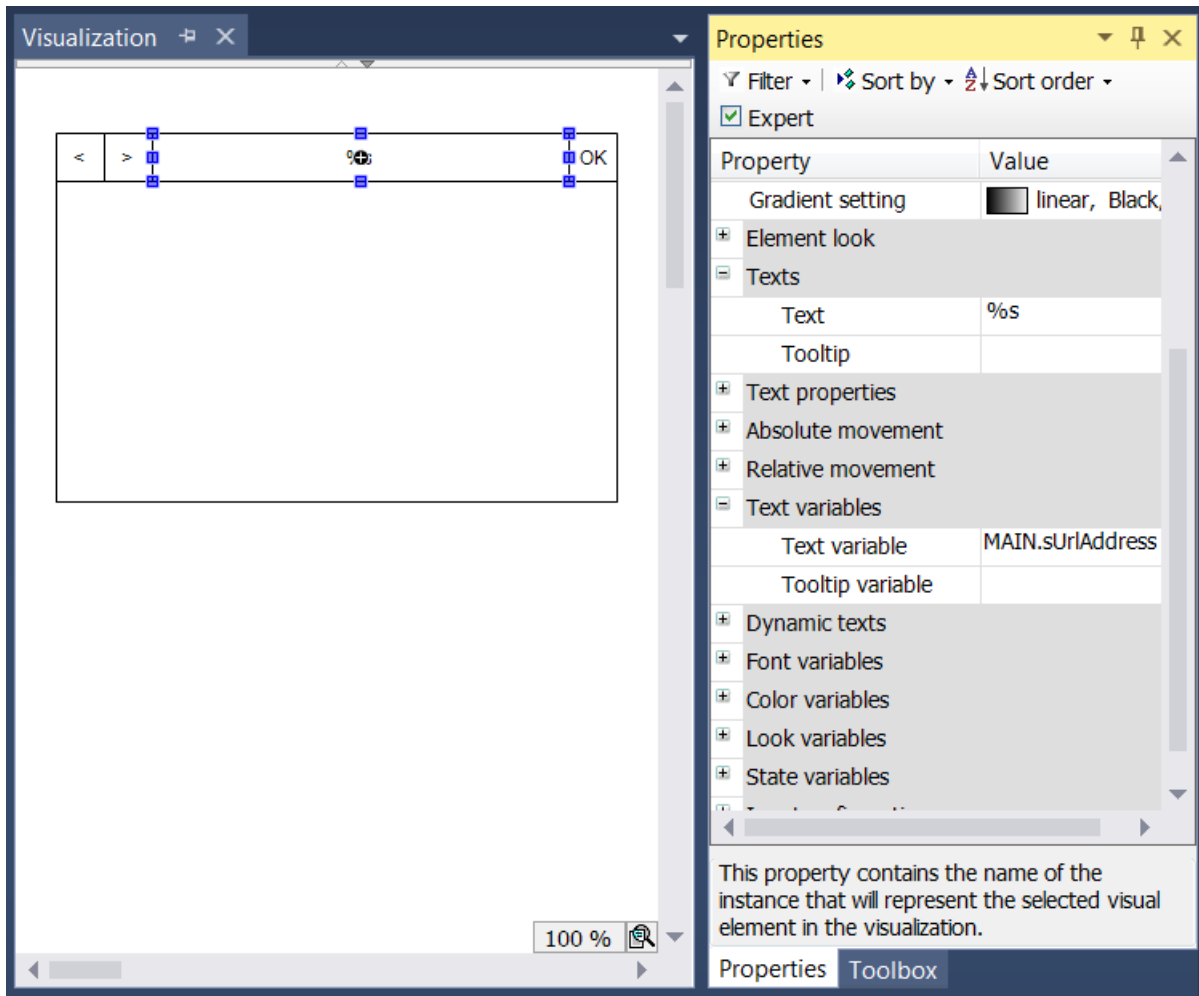
```
1 PROGRAM MAIN
2 VAR
3     sUrlAddress : STRING;
4     bShow       : BOOL;
5     bBack       : BOOL;
6     bForward    : BOOL;
7 END_VAR
```

The bottom part of the window is a visualization area, currently empty, with a yellow header bar.

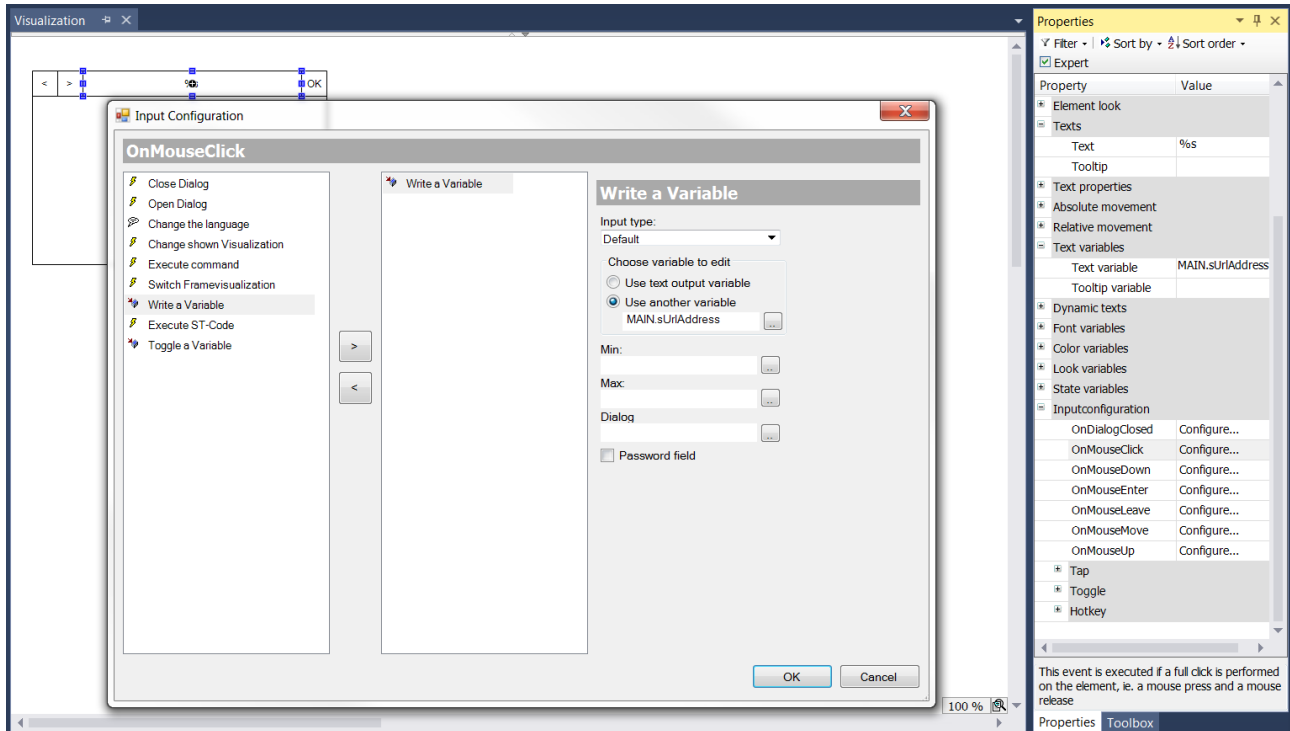
In order to be able to change the variables from the visualization declared in the program "MAIN", first three rectangle elements are dragged to the visualization. The first element with the text "<" is linked in the input configuration with "bBack" for the switchover function, the second with the text ">" is linked with "bForward" and the third with the text "OK" is linked with "bShow".



A fourth rectangle element is then added, in order to be able to change the URL address of the web browser. The placeholder "%s" is entered in the property "Text" and "sUrlAddress" is entered in "Text variable", so that the element can display the address.

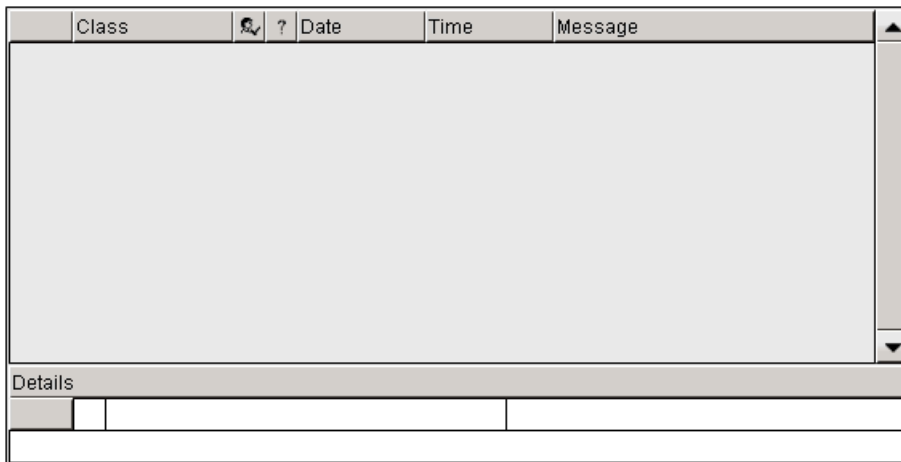


The "OnClick" event is used in the input configuration, in order to be able to change the URL address in the visualization. Here, a "Write variable" action is added and linked to the variable "sUrlAddress".



15.8.6.5 Event table

The event table element can be used to display TcEventLogger messages in the form of a table on a visualization page. The messages are read via the function block "FB_AdsReadEvents [▶ 597]" and forwarded to the visualization element via an array. An example of the configuration of the element can be found in the section "Configuration of the event table [▶ 598]". This element is available from build 4020.0.



Properties editor

The properties of a visualization element - except alignment and order [▶ 377] - can all be configured in the properties editor [▶ 385]. By default, this editor opens next to the visualization editor, or it can be opened explicitly via the "Properties" command (which can be found in the View menu as standard).

A property can be modified by editing the field "Value". To this end, an input field, a selection list, a dialog or checkbox that can be activated is provided in this field, depending on the element type. The value field opens

- after a double-click,
- after a single click in a selected field,
- via the space bar, if the field was already selected.

If a variable is assigned,

- simply enter its name.
- Use the button to open the input assistant for selecting a variable. The Variables category lists all variables that have already been defined in the project.

Working in the list of properties can be made easier with the aid of default, sorting and filter functions.

Element properties

All element properties and their descriptions are listed below.

Element name	The element name can be changed. Standard name is "GenElemInst_x". "x" stands for a sequential number.
Element type	The element type is entered here. For three element groups it is possible to switch between the corresponding elements by changing the element type: <ul style="list-style-type: none"> • <u>rectangle, rounded rectangle and ellipse [▶ 475]</u> • <u>Polygon, polyline or Bézier curve [▶ 490]</u> • <u>dip switch, power switch, push switch, push switch with LED, rocker switch [▶ 532]</u>

Position

Here you can define the position (X/Y coordinates) and size (width and height) of the element in pixels. The origin is in the top left corner of the window. The positive x-axis is on the right, the positive y-axis runs downwards. If the values edited, the displayed element is simultaneously modified in the [visualization editor](#) [► 376].

X	Horizontal position in pixels – X=0 is the left edge of the window.
Y	Vertical position in pixels – Y=0 is the upper edge of the window.
Width	Width of the element in pixels
Height	Height of the element in pixels

Columns

The element TcEventTable contains a table with the following seven columns:

- Index: The index indicates the message numbers in the order in which they occurred.
- Class: The class describes the message type. It is defined when a message is created.
- Acknowledgment state: A message can be created with obligatory acknowledgment. The acknowledgment can take place in the program code or in the element. The state indicating whether the element is (still) to be acknowledged is displayed in this column.
- Reset state: Once a message has been enabled, it can be reset in the program code. This row indicates whether the message was already reset.
- Date, Time: The two columns 'Date' and 'Time' show the date and time at which the message has occurred.
- Message: The actual message text is displayed in the last column.

Columns	Setting options for the seven columns
Row height	Row height in pixels
Scrollbar width	Width of the scrollbar in pixels
Sorting order	Here you can set the order in which the messages are displayed in the element: <ul style="list-style-type: none"> • Newest first • Oldest first

Column

Column width	Column width in pixels
Text properties	Here you can modify the text properties for the column: <ul style="list-style-type: none"> • Horizontal alignment • Vertical alignment • Font • Font color

Detail properties

In addition to the table, in which the messages are displayed, the element TcEventTable has a detail field. If one of the messages in the table is selected at runtime, further information about the message is displayed

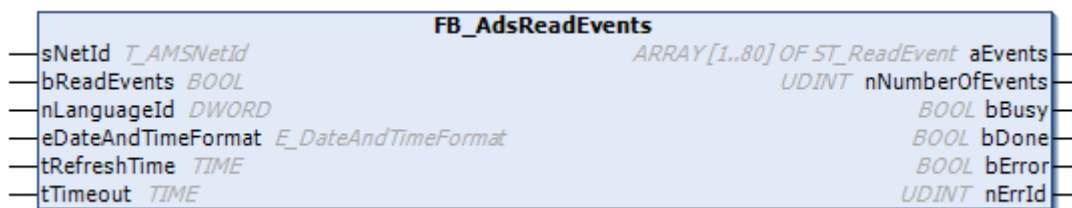
in the detail field. The acknowledgement button  can be used to acknowledge this message.

General text properties	Here you can set the text properties for all detail cells apart from the message cell: <ul style="list-style-type: none"> • Horizontal alignment • Vertical alignment • Font • Font color
Message text properties	Here you can set the text properties for the message cell: <ul style="list-style-type: none"> • Horizontal alignment • Vertical alignment • Font • Font color

Auxiliary setting

Message data array	Here, the output variable "aEvents" of the function block "FB_AdsEventReader [▶ 597]" has to be assigned, in which the active messages of TcEventLogger are stored.
---------------------------	---

15.8.6.5.1 FB_AdsReadEvents



The function block queries the active messages of the EventLogger via ADS and makes them available in the form of an array `aEvents`. To display the messages in the visualization element `Event table [▶ 595]`, the array `aEvents` has to be entered in its property `Message data array [▶ 597]`.

Messages with a text length less than or equal to 255 characters can be output in full at the output. Messages with a text length greater than 255 characters and less than or equal to 1023 characters are output with truncated text. Messages with a text length greater than 1023 characters cannot be output and the function block returns an error.

VAR_INPUT

```

VAR_INPUT
  sNetId      : T_AMSNetId;
  bReadEvents : BOOL;
  nLanguageId : DWORD;
  eDateAndTimeFormat : E_DateAndTimeFormat;
  tRefreshTime : TIME;
  tTimeout    : TIME;
END_VAR

```

sNetId: AmsNetID of the device, from which the messages of the EventLogger are to be queried. If the messages are to be read locally, an empty string can be specified.

bReadEvents: The input can be used to enable reading of the messages. When the enable is reset, the error outputs (`bError` and `nErrId`) are also reset.

nLanguageId: (Language Id) Defines the language in which message texts are displayed.

eDateAndTimeFormat: Defines the timestamp format. The available options are:

- `de_De` – German spelling: `dd.MM.yyyy hh:mm:ss (24 h)`
- `en_GB` – British spelling: `dd/MM/yyyy hh:mm:ss (12 h)`

- en_US – American spelling: MM/dd/yyyy hh:mm:ss (12 h)

tRefreshTime: Defines the time interval, after which the message query is repeated.

tTimeout: Defines the time interval, after which a timeout error is triggered.

VAR_OUTPUT

```
VAR_OUTPUT
  aEvents      : ARRAY[1..80] OF ST_ReadEvent;
  nNumberOfEvents : UDINT;
  bBusy        : BOOL;
  bDone        : BOOL;
  bError       : BOOL;
  nErrorId     : UDINT;
END_VAR
```

aEvents: The function block uses this array to make the read messages available. The array can store a maximum of 80 messages. (see ST_ReadEvent)

nNumberOfEvents: Indicates how many messages are currently stored in the array aEvents.

bBusy: Indicates whether the function block is currently busy.

bDone: TRUE, if the function block is not busy at present, but has carried out at least one operation.

bError: Indicates whether an error has occurred.

nErrorId: Indicates the error ID.

Requirements

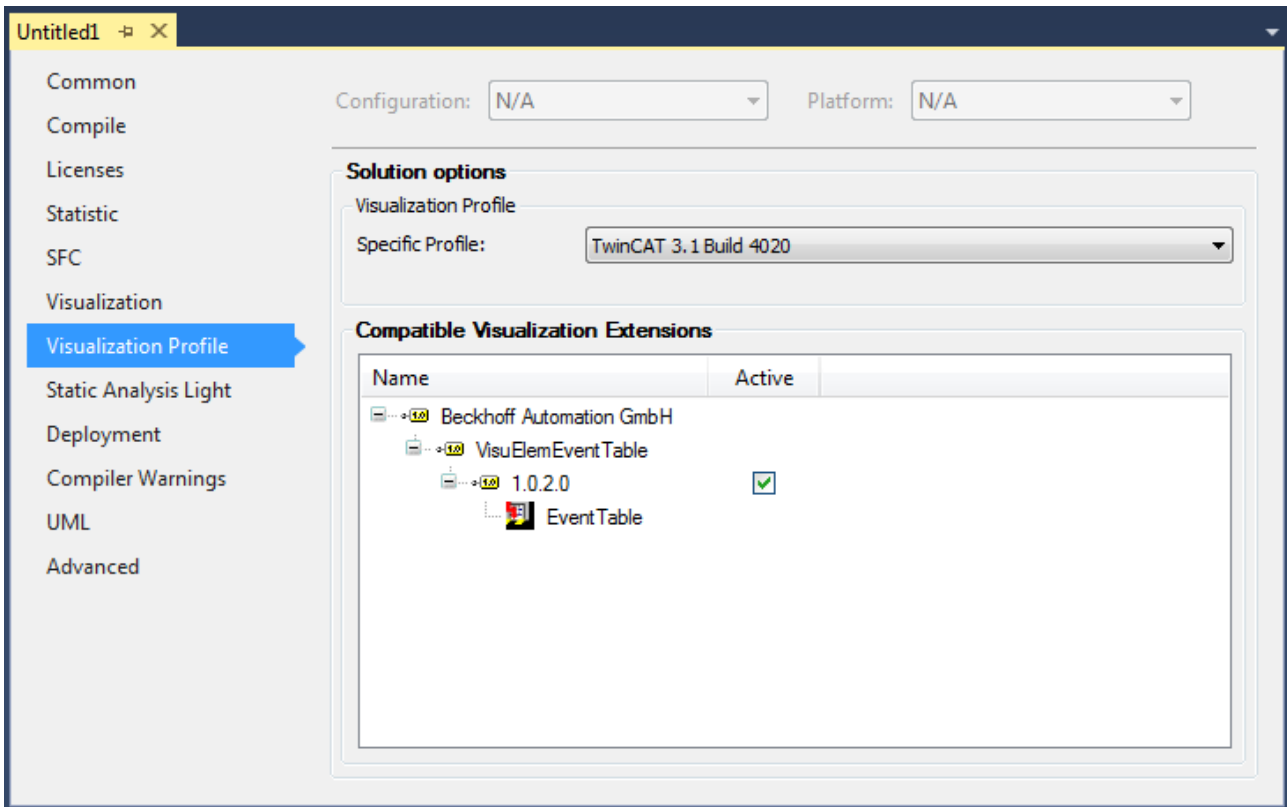
Development environment	Target platform	PLC libraries to be integrated (category group)
TwinCAT v3.1.0	PC or CX (x86, x64, ARM)	Tc2_Uilities (System)

15.8.6.5.2 Configuration of the event table

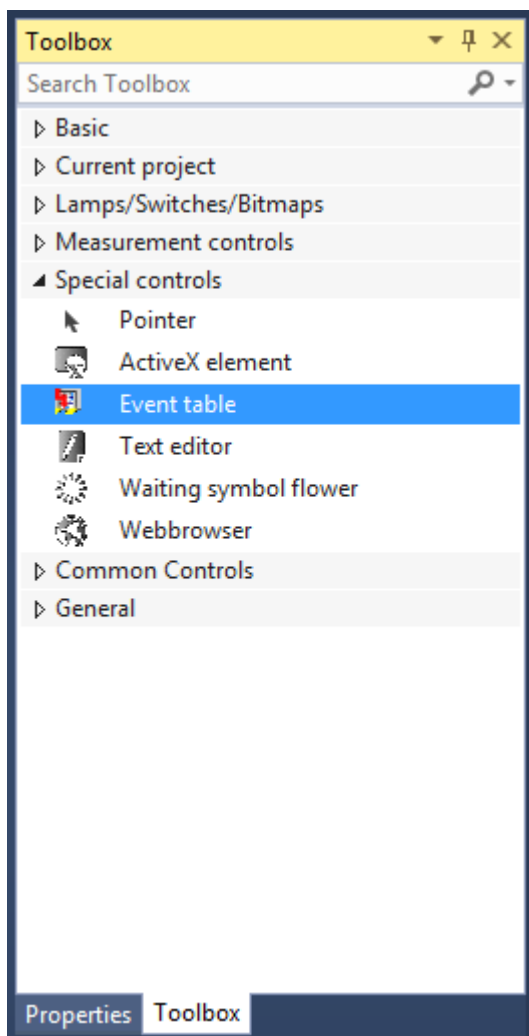
The visualization element "event table" works in combination with the function block "[FB_AdsReadEvents \[► 597\]](#)". The function block is therefore added to the PLC program first. It is included in the library "Tc2_Uilities". In this example the function block is declared and called in the program "MAIN". Since the messages of the local TcEventLogger are to be read, an empty string can be entered under the input "sNetId".

```
PROGRAM MAIN
VAR
  fbAdsReadEvents : FB_AdsReadEvents;
  bReadEvents : BOOL;
END_VAR
fbAdsReadEvents ( sNetId := , \,
  bReadEvents := bReadEvents,
  nLanguageId := 1031,
  eDateAndTimeFormat := E_DateAndTimeFormat.de_DE,
  tRefreshTime := T#1S,
  tTimeout := T#5s);
```

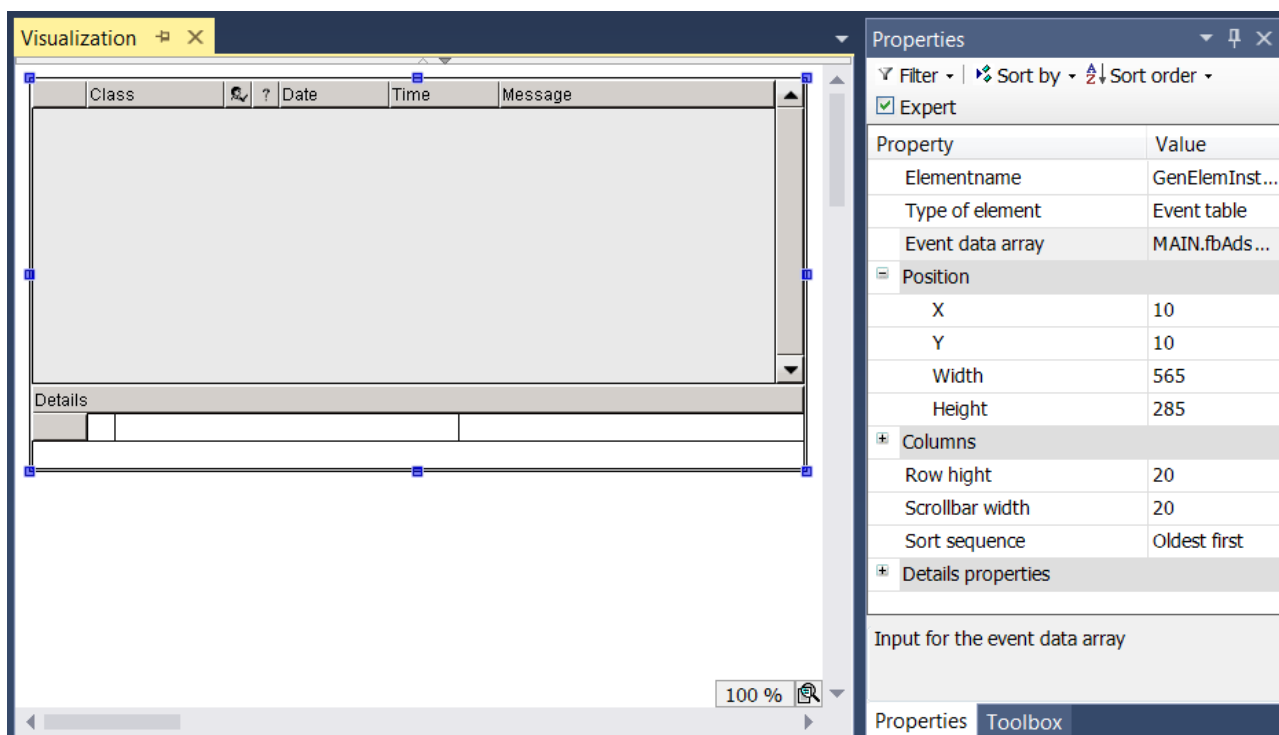
In order to be able to add the visualization element "event table" on a visualization page, the corresponding extension for the element must be selected in the [PLC project settings \[► 402\]](#) under the category "Visualization Profile".



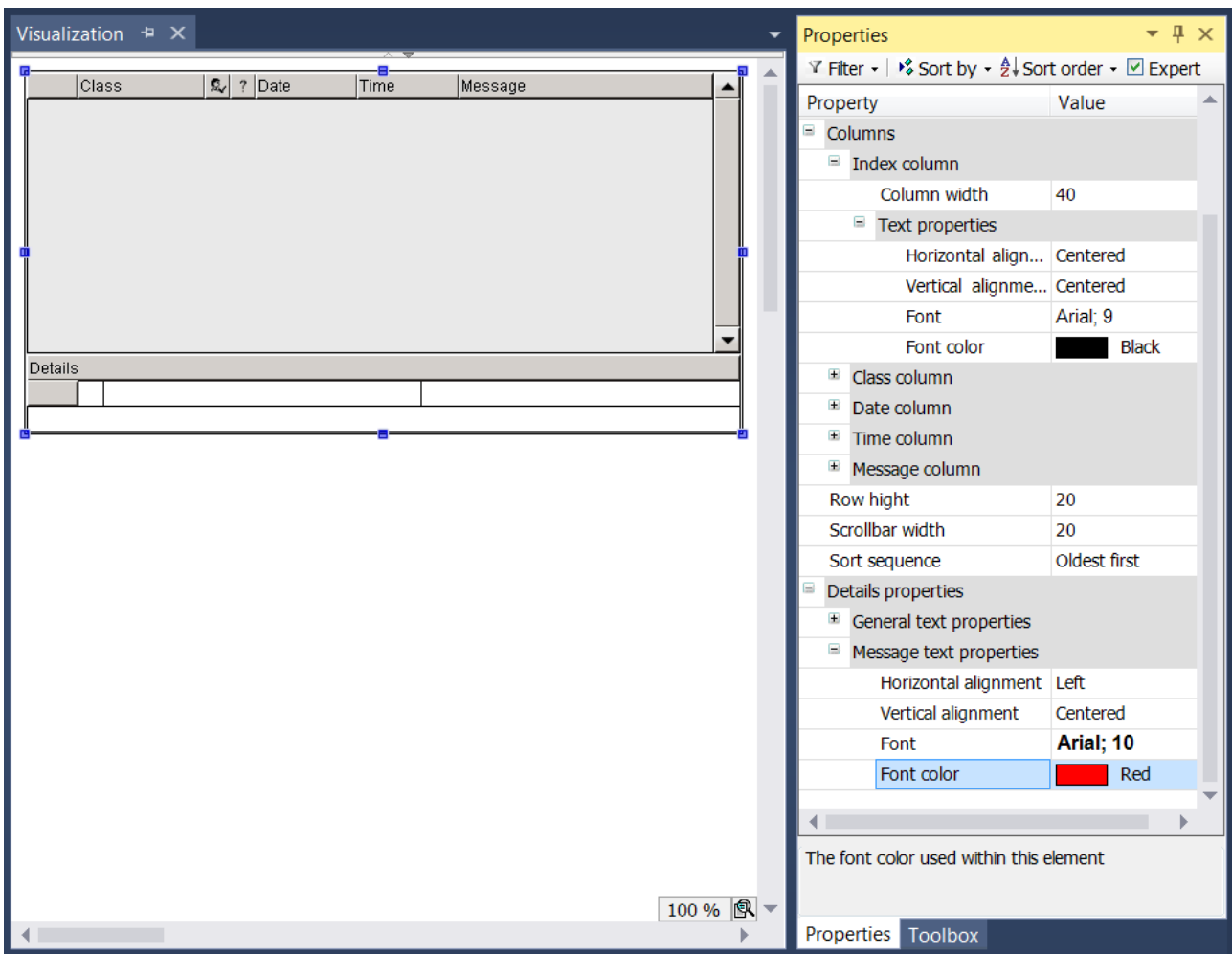
After activation of this setting the TwinCAT project has to be restarted once. The element is then available in the [toolbox](#) [▶ 384] in the category "[special control elements](#) [▶ 580]" and can be used on the visualization page.



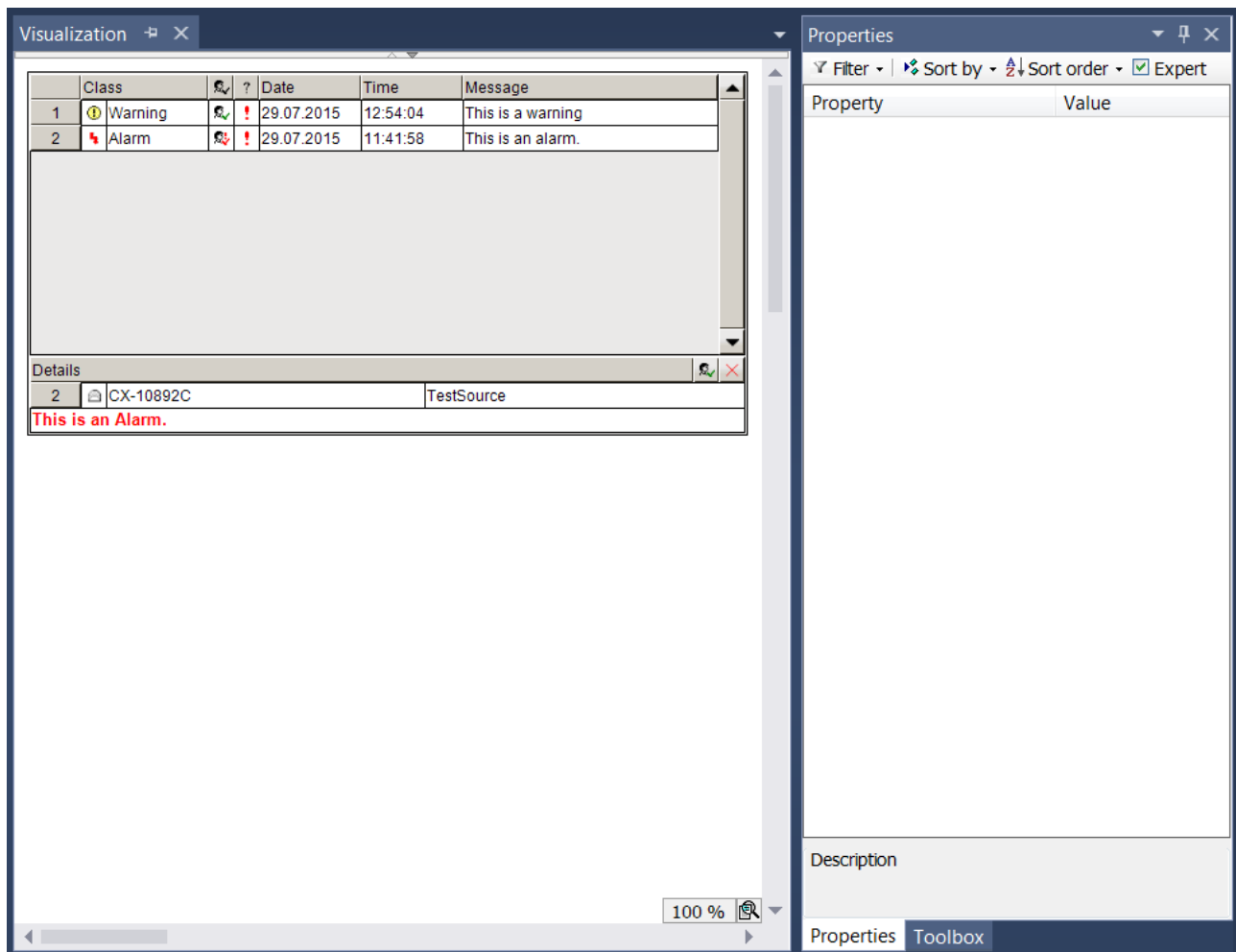
Once the element has been added to the visualization page, the array "aEvents" of the instance "fbAdsReadEvents", in which the messages are stored, is entered on the properties, and the size is set to 565x285 pixels.



To center the text in the index column, change the text property "Horizontal alignment" to "Centered" in the category "Columns" → "Index column". Moreover, the font size and color of the message text is adjusted in the "Detail properties", in order to distinguish it from the other information.



At runtime the visualization element looks as follows, if two example messages were triggered.



15.9 Visualization variants

A distinction is made between three different visualization variants in the PLC project:

- [Integrated visualization \[▶ 602\]](#) – visualization runs in the development environment of TwinCAT on the programming system
- [PLC HMI \[▶ 603\]](#) – visualization runs without development environment on the control computer or a third computer
- [PLC HMI Web \[▶ 608\]](#) – visualization runs in a browser

Despite the three different variants, engineering is necessary only once, as a result of which the look and feel is identical in all visualizations. PLC HMI and PLC HMI Web can be enabled by simply adding [TargetVisualization \[▶ 606\]](#) and [WebVisualization \[▶ 609\]](#) objects. The availabilities of the individual [Visualization elements \[▶ 404\]](#) and functions in the different variants can be found in the section "[Availability \[▶ 610\]](#)"

15.9.1 Integrated visualization

For diagnostic purposes it may be desirable to run a visualization only within the programming system, without having to load visualization code on the controller. This integrated visualization is used automatically, if no "[TargetVisualization \[▶ 606\]](#)" or "[WebVisualization \[▶ 609\]](#)" client object was added under the [Visualization Manager \[▶ 387\]](#). In this case no visualization code is generated and loaded on the controller. However, this involves some restrictions, which are listed below.

Restrictions for expressions, monitoring

The diagnostic visualization supports only expressions, which can be handled by the monitoring mechanism of the programming system. These are:

- normal variable access such as MAIN.fbTest.nCounter
- Complex access, as listed below:
 - Access to an array of scalar data types, whereby one variable is used as index (a[i])
 - Access to an array of complex data types (structures, function blocks, arrays), whereby one variable is used as index (a[i].x)
 - Access to a multi-dimensional array of all data types, with one or several variable indices (a[i, 1, j].x)
 - Access to an array with constant index (a[3])
 - Access as described above, in which simple operators are used for the calculations within the index bracket (a[i + 3])
 - Nested combinations of the complex expressions listed above (a[i + 4 * j].alnner[j * 3].x)
- Operators supported in index calculations: +, -, *, /, MOD
- Pointer monitoring such as p^.x
- Methods or function calls are not supported, except the following:
 - all default string functions
 - all type conversion functions, such as INT_TO_DWORD
 - all operators such as SEL, MIN, ...

Restrictions for inputs

Within the input action "Run ST code" only one list of assignments is supported.

Example:

```
PLC_PRG.n := 20 * PLC_PRG.m;
// nicht erlaubt
IF PLC_PRG.n < MAX_COUNT THEN
PLC_PRG.n := PLC_PRG.n + 1;
END_IF
//statt dessen folgendes verwenden:
PLC_PRG.n := MIN(MAX_COUNT, PLC_PRG.n + 1);
```



If a list of assignments is used, the value on the left is not assigned until the next cycle. Immediate processing in the next line is not possible.

Visualization interface

The type "Interface" may not be used within the interface definition for a visualization.

15.9.2 PLC HMI


The PLC HMI is an extension of the runtime system and enables the visualization to be executed on the control computer or a third computer without a development environment. The visualization code is created based on the existing visualization objects and downloaded to the control computer. Avoiding the use of the development environment results in significant memory savings. This can be useful for small computers.

The following topics are described below:

- [Commissioning the PLC HMI \[► 604\]](#)
- [Remote operation of a PLC HMI Client \[► 606\]](#)
- [Editor of the TargetVisualization object \[► 606\]](#)

Commissioning the PLC HMI

Step 1: Enable the PLC HMI

The object "TargetVisualization" () enables the PLC HMI. It can be added to the object "Visualization Manager" in the PLC project tree via the context menu command **Add > TargetVisualization** (see also PLC documentation: Creating a visualization > [Visualization object](#)).

With the TargetVisualization object a visualization task "VISU_TASK" is created in the Solution and a reference to this task in the PLC project. The reference is used to call the visualization code. Therefore, you have to reactivate the configuration after adding the object.

● Deleting a TargetVisualization object

i If you delete a TargetVisualization object and have not added an additional WebVisualization object, you have to delete the task "VISU_TASK" under **System > Tasks** in the TwinCAT project tree. This task is not required in the integrated visualization. (See also [Editor of the object WebVisualization](#) and [Integrated visualization](#))

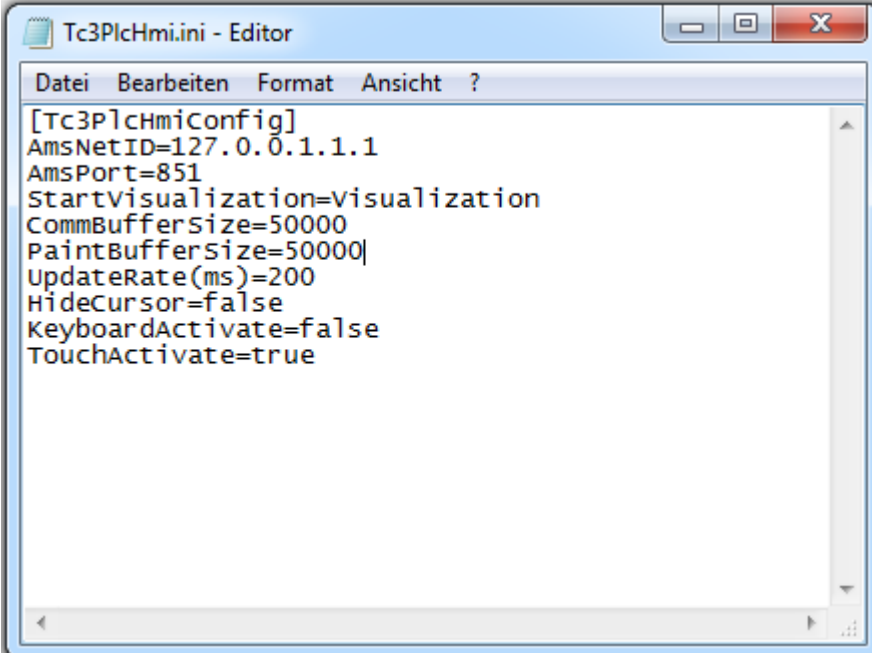
Step 2: Configure the PLC HMI Client

i Step 2 is only necessary if you are using a Build <4022.0 or if you want to start a PLC HMI Client with a remote connection to the runtime device. From Build 4022.0 or higher, the .ini file is automatically generated and updated in the folder *C:\TwinCAT\3.1\Boot\Plc*. From Build 4026.0 or higher, the .ini file is automatically generated and updated in the folder *C:\ProgramData\Beckhoff\TwinCAT\3.1\Boot\Plc*.

In order to establish the connection between the client and the device on which the corresponding visualization code is executed, you must adapt the Tc3PlcHmi.ini file.

The .ini file is available for Builds <4022.0 in the folder *C:\TwinCAT\3.1\Components\Plc\Tc3PlcHmi*, for Builds >=4022.0 in the folder *C:\TwinCAT\3.1\Boot\Plc* and for Builds >=4026.0 in the folder *C:\Program Files (x86)\Beckhoff\TwinCAT\3.1\Components\Plc\Tc3PlcHmi*.

Example of a .ini file:



```

Tc3PlcHmi.ini - Editor
Datei Bearbeiten Format Ansicht ?
[Tc3PlcHmiConfig]
AmsNetID=127.0.0.1.1.1
AmsPort=851
StartVisualization=visualization
CommBufferSize=50000
PaintBufferSize=50000
UpdateRate(ms)=200
HideCursor=false
KeyboardActivate=false
TouchActivate=true
  
```


AMSNETID	AmsNetID of the device, on which the visualization code is executed. Preset: 127.0.0.1.1.1
AmsPort	AmsPort of PLC project, to which the visualization belongs. Preset: 851
StartVisualization	Name of the visualization object to be opened as start page. Preset: Visualization
CommBufferSize	Memory size in bytes that the visualization allocates for this PLC HMI Client and uses for the communication. Preset: 50000
PaintBufferSize	Memory size in bytes that the visualization allocates for this PLC HMI Client and uses for the drawing actions. Preset: 50000
UpdateRate(ms)	Update rate in milliseconds, at which the client data are queried again. Preset: 200
HideCursor	Setting through which the cursor can be hidden. Preset: false
KeyboardActivate	Setting through which input via a hardware keyboard is enabled. A software keyboard is used automatically if this setting is disabled. Preset: false
TouchActivate	Setting through which touch-based input is enabled. Preset: true

Step 3: Set PLC HMI as startup application



Step 3 is only necessary if you are using a Build <4024.0 or if you want to start a PLC HMI Client with a remote connection to the runtime device. From Build 4024 or higher, the PLC HMI Client is automatically started locally on the runtime device.

If PLC HMI is to start automatically when the computer is booted with the boot project, there must be a link to the Tc3PlcHmi.exe application in the *StartUp* folder.

Execute the following steps to do this:

1. Open the directory *C:\TwinCAT\3.1\Target\StartUp*.
2. Add a new link via the context menu command **New**.
3. Enter *C:\TwinCAT\3.1\Components\Plc\Tc3PlcHmi\Tc3PlcHmi.exe* as storage location.
4. Confirm this dialog and the following dialog.

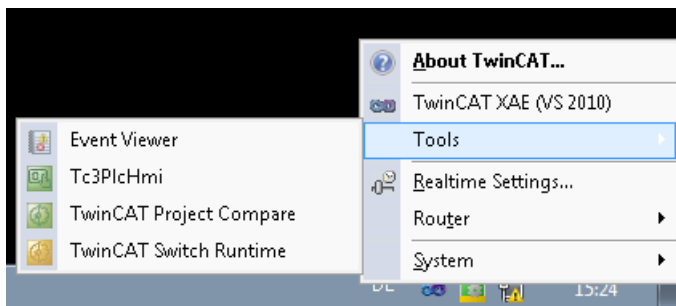
Execute the following steps for Beckhoff CE devices:

1. Start the Beckhoff Startup Manager under **Start > StartMan**.
2. Add a new item via the **New** button.
3. Give the item the name "Tc3PlcHmi" and select the type "ShellCommand".
4. Confirm the dialog.
5. Under the **Startup Options**, select "Autostart" and enter a time under **Delay** in order to open the client only when the PLC project has already been started.
6. Switch to the **Shell Command** tab.
7. In the field **Enter Shell command**, enter "*Hard Disk\TwinCAT\3.1\Components\Plc\Tc3PlcHmi\X.exe*". Replace the "X" with the name of the Client Exe that is stored under the specified path. This may differ between ARM and ATOM devices, for example.
8. Confirm the dialog.

Step 4: Start the PLC HMI Client

i Step 4 is only necessary if you are using a Build <4024.0 or if you want to start a PLC HMI Client with a remote connection to the runtime device. From build 4024 or higher, the PLC HMI Client is automatically started locally on the runtime device.

A PLC HMI Client is started with the aid of the Tc3PlcHmi.exe application. This is located in the directory `C:\TwinCAT\3.1\Components\Plc\Tc3PlcHmi`, but can also be linked to any desired location. If you create a link in the directory `C:\TwinCAT\3.1\Target\StartMenuAdmin\Tools` you can start the application via the TwinCAT icon in the context menu under **Tools**.



If the development PC is connected, the visualization can also be displayed in the development environment. However, it is not equivalent to an integrated visualization, but is also based on a PLC HMI Client.

For Beckhoff CE devices you have to activate a setting in the visualization manager before starting the client; this setting enables all image files in the svg format to be automatically converted to the bmp format. This step is required, because under CE only image files in bmp format are supported in the PLC HMI Client. Both image file formats are nevertheless loaded on the target system, since a PLC HMI Web Client continues to use the svg format. The PLC HMI Client for CE can be found in directory `\Hard Disk\TwinCAT\3.1\Components\Plc\Tc3PlcHmi`.

See also:


- PLC documentation: Creating a visualization > Visualization Manager > [Settings](#)
- PLC documentation: Creating a visualization > Visualization variants > [Integrated visualization](#)
- Documentation [TC3 PLC HMI Web](#)

Remote operation of a PLC HMI Client

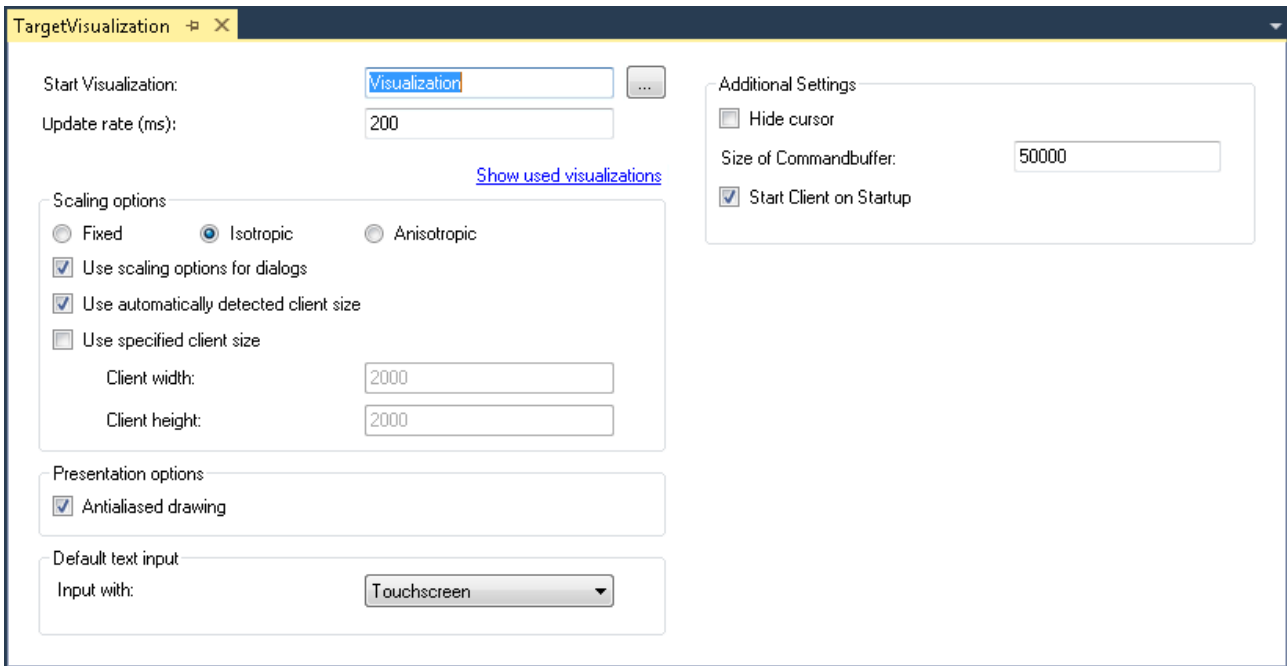
A PLC HMI Client can also be operated remotely on a third computer, which is neither the development computer nor the control computer. To do this, the following requirements must be met:

- A TwinCAT 3 Build 4018.0 ADS or later is installed on the system.
- ADS communication is established with the control computer on which the visualization code is executed (**TwinCAT Icon > Router > Edit Routes > Add...**).
- The Tc3PlcHmi folder has been copied from the development or control computer to the third system. The path for the folder must be added manually.
- The Tc3PlcHmi.ini file was adjusted on the system, on which the client is to run.

Editor of the TargetVisualization object

The "TargetVisualization" object (), which you can add in the PLC project tree below the "Visualization Manager" object, enables the PLC HMI and contains its settings. Double-click on the object in order to edit the settings in an editor window.

i The settings in the object "TargetVisualization" are adopted automatically into the .ini file from build 4022.0 onwards. If you wish to use an older build or start a PLC HMI Client with a remote connection to the runtime device, you have to make the changes to the settings in the .ini file manually.



Start visualization	Name of the visualization object that is to be opened as the first page when starting the PLC HMI. A visualization object is already entered here by default. The input assistant can be used to select a different visualization object. If the PLC project contains only one visualization object, this is automatically used as start visualization.
Update rate (ms)	The update rate in milliseconds, with which the data in the PLC HMI is updated.
Show used Visualizations	Button for opening the standard dialog of the Visualization Manager: Here you can select the visualizations that are to be used for the PLC HMI. (See also PLC documentation: Creating a visualization > Visualization Manager > Visualizations [▶ 393])

Scaling options

Fixed	The size of the visualization is retained, irrespective of the screen size.
Isotropic	The size of the visualization depends on the size of the screen. The visualization retains its proportions.
Anisotropic	The size of the visualization depends on the size of the screen. The visualization does not retain its proportions.
Using scaling options for dialogs	The dialogs, also keypad and numpad, are scaled with the same scaling factor as the visualization. This is advantageous if a dialog was created to match the visualization.
Use automatically determined client size	The PLC HMI fills the client screen.
Use specified client size	The PLC HMI fills the screen area determined by the following dimensions. <ul style="list-style-type: none"> • Client height: height in pixels • Client width: width in pixels

Presentation options

Characters with antialiasing	Activate this option, if antialiasing is to be used when the visualizations are drawn in the visualization editor window of the programming system. (Offline or online)
------------------------------	---

Standard text input

This setting is only then active if the input type “Standard” is selected in the input configuration of the visualization element. In this case, the default text entries defined in the Visualization Manager are used.

Touchscreen	Select this option if the target device is operated with a touch screen by default.
Keyboard	Select this option if the target device is operated with a keyboard by default.

Advanced Settings

Hide mouse pointer	Setting through which the cursor can be hidden.
Size of the command buffer	Memory size in bytes that the visualization allocates for this PLC HMI Client and uses for the communication.
Start client on startup	The PLC HMI client is automatically started locally on the runtime system.

15.9.3 PLC HMI Web

PLC HMI Web enables the visualization to be displayed in any web browser. It is realized as a Java script, which queries the display information from the web server. Only changes in the display are transferred cyclically. When a visualization project is downloaded, all files required for the PLC HMI Web up to <TC3.1.4026.0 are transferred to directory *C:\TwinCAT\3.1\Boot\Plc\Port_851\Visu* and from >=TC3.1.4026.0 to the directory *C:\ProgramData\Beckhoff\TwinCAT\3.1\Boot\Plc\Port_851\Visu*. This includes the Java script, the basic HTML page (HTM file) for the visualization, and all images required in the visualization.



PLC HMI Web can currently only be configured for PLC projects that can be reached via port 851.

The following topics are described below:

- [Requirements \[▶ 608\]](#)
- [Commissioning of the PLC HMI Web \[▶ 608\]](#)
- [Editor of the WebVisualization object \[▶ 609\]](#)

Requirements

- On the server side the web server must be configured accordingly.
- On the client side, as a minimum Microsoft Internet Explorer 10 or the latest version of Mozilla Firefox, Google Chrome or Safari must be available.



Data security violations

In order to minimize the risk of data security violations, the following organizational and technical measures are recommended for the system, on which your applications run:


- Exposure of the PLC and the control network to open networks and the Internet should be avoided, as far as possible.
- Use additional data link layers such as a VPN for remote access and install firewall mechanisms for protection.
- Restrict access to authorized persons, change any default passwords during first commissioning and then at regular intervals.

Commissioning the PLC HMI Web

Step 1: Configure Internet Information Services (IIS)

The PLC HMI Web uses the Microsoft IIS as web server. The IIS has to be configured accordingly. The configuration is handled by the [TF1810 | TC3 PLC HMI Web](#) installation, which is available for download from the Beckhoff website.

Step 2: Enable PLC HMI Web

The "WebVisualization" object () enables the PLC HMI Web. You add it to the "Visualization Manager" object in the PLC project tree via the context menu command **Add > WebVisualization** (see also PLC documentation: [Creating a visualization > Visualization object](#)).

With the WebVisualization object a visualization task "VISU_TASK" is created in the Solution and a reference to this task in the project. The reference is used to call the visualization code. Therefore, you have to reactivate the configuration after adding the object.

Deleting a WebVisualization object

I If you delete a WebVisualization object and have not added an additional TargetVisualization object, you have to delete the task "VISU_TASK" under **System > Tasks** in the TwinCAT project tree. This task is not required in the integrated visualization. (See also TF1800: [Editor of the TargetVisualization object](#) and PLC: [Integrated visualization](#))

Step 3: Call PLC HMI Web

In order to call the start page of the visualization, enter the following address in the web browser: `https://device_name/Tc3PlcHmiWeb/Port_851/Visu/webvisu.htm`


Example: `https://localhost/Tc3PlcHmiWeb/Port_851/Visu/webvisu.htm`

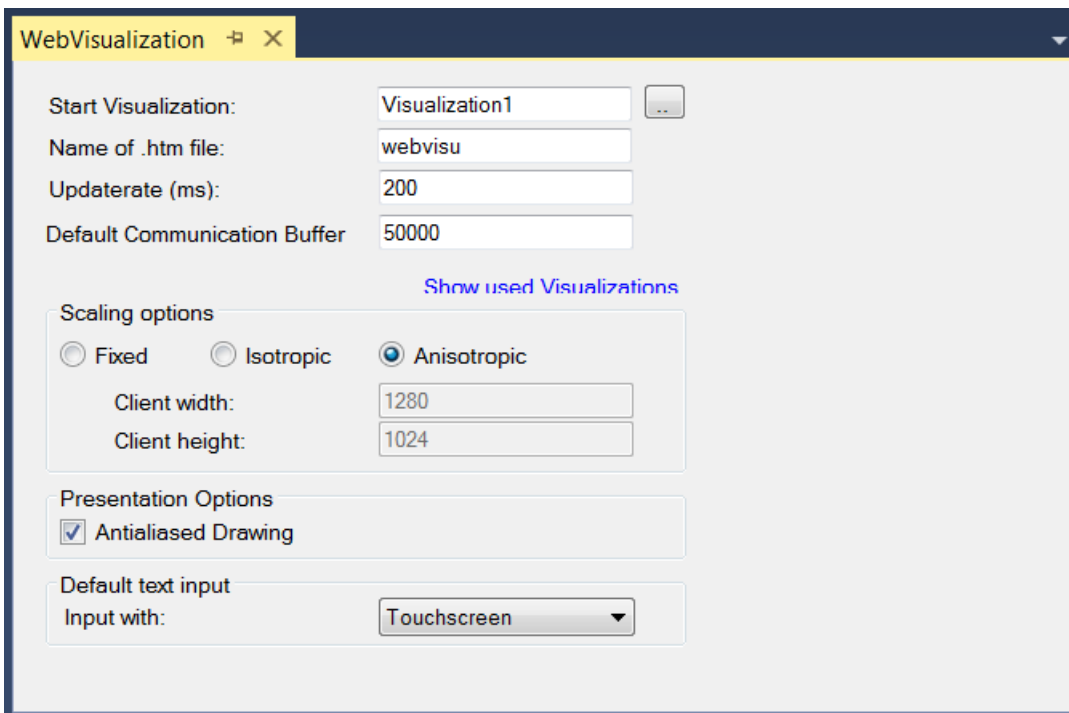
"webvisu" is the HTML start page of the visualization defined in the PLC HMI Web settings. After the call this is used to display the start visualization, which is also defined in the manager, in the browser. The visualization can then be operated in the browser.

Optionally you can give the PLC HMI Web a name on calling it in order to be able to address it specifically in the application later on. To do this, enter the parameter `ClientName=<Name>` after the URL.

Example: `https://localhost/Tc3PlcHmiWeb/Port_851/Visu/webvisu.htm?Clientname=V_ClientXY`

Editor of the WebVisualization object

The "WebVisualization" object (), which you can add in the PLC project tree below the "Visualization Manager" object, enables the PLC HMI Web and contains the settings for the web visualization. Double-click on the object in order to edit the settings in an editor window.



Start Visualization	Name of the visualization to be displayed automatically when the PLC HMI Web is started. "Visualization" is to be entered here as standard. The input assistant can be used to select a different visualization.
Name of .htm file	Name of the basic HTML page of the visualization, which must also be entered as the address in the web browser. Example: https://localhost/Tc3PlcHmiWeb/Port_851/Visu/webvisu.htm
Updaterate (ms)	Update rate in milliseconds, with which the data in the web browser are updated.
Default Communication Buffer	Size of the communication buffer in bytes. Specifies the maximum available memory for the data transfer between web client and web browser.
Show used Visualizations	Button for opening the standard dialog of the Visualization Manager: Here you can select the visualizations that are to be used for the PLC HMI Web. (See also PLC documentation: Creating a visualization > Visualization Manager > Visualizations [► 393])

Scaling options

Fixed	The size of the visualization is retained, irrespective of the size of the browser window.
Isotropic	The size of the visualization depends on the size of the browser window. The visualization retains its proportions, however.
Anisotropic	The size of the visualization depends on the size of the browser window. The visualization does not retain its proportions.
Client size	The display size of PLC HMI Web is defined through the following settings: <ul style="list-style-type: none"> • Client height: height in pixels • Client width: width in pixels

Presentation options

Antialiased Drawing	Activate this option, if antialiasing is to be used when the visualizations are drawn in the visualization editor window of the programming system. (Offline or online)
---------------------	---

Default text input

This setting is only then active if the input type "Standard" is selected in the input configuration of the visualization element. In this case, the default text entries defined in the Visualization Manager are used.

Touchscreen	Select this option if the web clients are operated with a touchscreen by default.
Keyboard	Select this option if the web clients are operated with a keyboard by default.

15.9.4 Availability

The following section describes the availability of the individual visualization elements and functionalities for the different visualization types.

Visualization elements

	Integrated visualization	PLC HMI	PLC HMI CE	PLC HMI Web
Common Controls [▶ 421]	✓	✓	✓	✓
Basic [▶ 475]	✓	✓	✓	✓
Lamps/Switches/Bitmaps [▶ 527]	✓	✓	✓	✓
Measuring devices [▶ 537]	✓	✓	✓	✓
Special controls [▶ 580]				
• Text Editor [▶ 582]	✗	✓	✓	✓
• ActiveX element [▶ 587]	✓	✗	✗	✗
• Webbrowser [▶ 589]	✓	✗	✗	✗
• TcEventTable [▶ 595]	✓	✓	✓	✓

Functionalities

	Integrated visualization	PLC HMI	PLC HMI CE	PLC HMI Web
User management [▶ 393]	✗	✓	✓	✓
CurrentVisu [▶ 388], CurrentLanguage [▶ 616]	✗	✓	✓	✓
Change the language [▶ 616]	✓	✓	✓	✓
Internal rotation [▶ 434]	✓	✓	✗	✓
Gradient Type [▶ 476]	✓	✓	✗	✓
Transparency [▶ 476]	✓	✓	✗	✓
Text format [▶ 422]	✓	✓	✗	✓

Image formats supported

	Integrated visualization	PLC HMI	PLC HMI CE	PLC HMI Web
SVG	✓	✓	✗	✓
BMP	✓	✓	✓	✓
JPG	✓	✓	✓	✓
PNG	✓	✓	✓	✓

The [visualization styles \[▶ 389\]](#) with a version higher than 3.0.0.0 use image files in SVG format internally. The function [Convert Images to \[▶ 389\]](#) can be enabled in the Visualization Manager in order to be able to use these styles with the PLC HMI CE. The function converts the image files into BMP or optionally PNG.

Both the converted and original image files are transmitted to the target system with the aid of the setting [Transfer both svg images and converted images \[▶ 389\]](#). A PLC HMI Web client then automatically uses the SVG variant and a PLC HMI CE client the BMP or PNG variant.

15.10 Application tips

15.10.1 Handling of visualization pages

The TwinCAT PLC Control concept of visualization references and placeholders is replaced by a similar one in TwinCAT 3.

Reference to another visualization

It is possible to add a [visualization page \[▶ 402\]](#) in another page and to reference it in this way. A [frame \[▶ 516\]](#) element should be used for this purpose. In this way a visualization page can be assembled from various other pages. A frame element can contain one or several references to visualization pages. These visualization pages are defined in the [frame selection \[▶ 525\]](#) dialog.

Interfaces for placeholders

Each [visualization page \[▶ 402\]](#) can provide an interface via the [interface editor \[▶ 378\]](#), in which the input variables can be defined in a way that is comparable to a function block. These input parameters act as placeholders. In an instance (reference) of the visualization page, they have to be replaced with values or expressions for the special application in the local object.

The replacement must be carried out in the [properties \[▶ 430\]](#) of the frame element, which integrates the visualization instance. Note that the input variables of a visualization instance must be assigned valid variables. If variables are changed in the interface editor, the dialog "[Updating the frame parameters \[▶ 526\]](#)" is opened for the placeholders of each instance. Here you can add or edit placeholders.

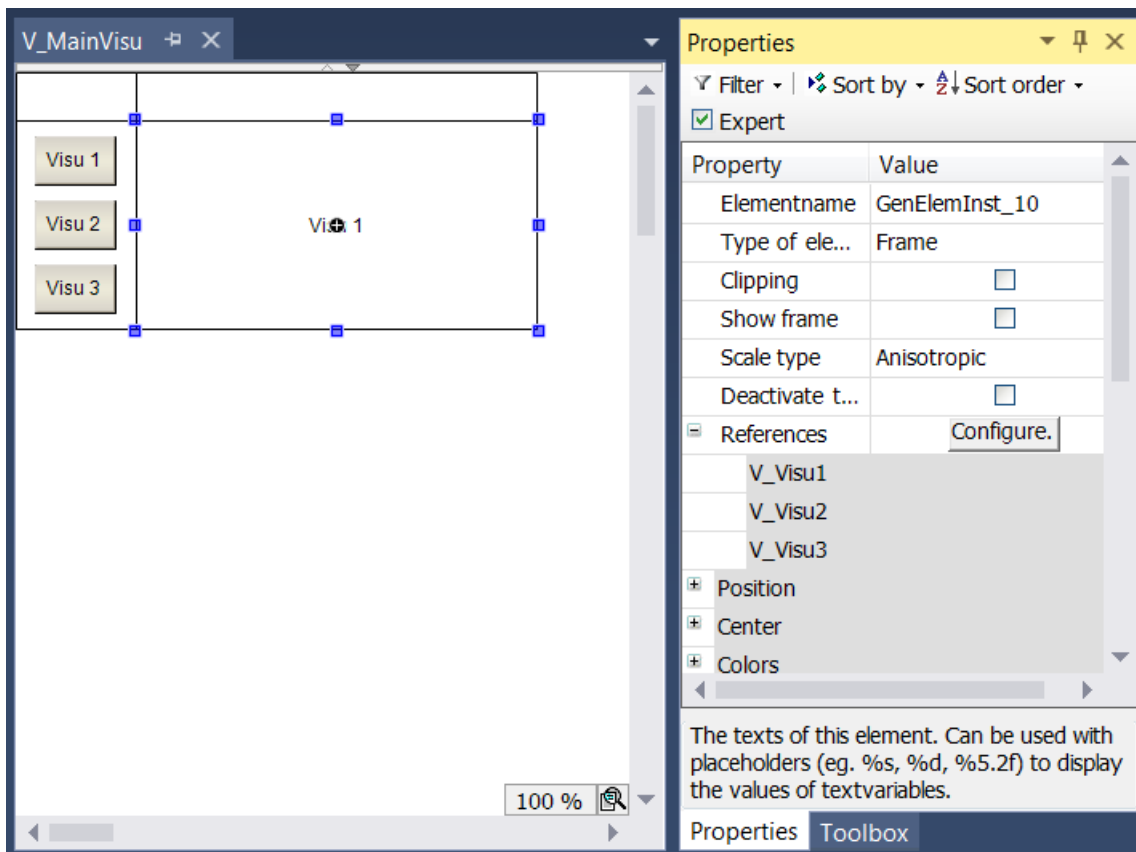
Switching between visualization pages within a frame

If a frame element contains several visualization references, user inputs for another visualization element can be configured such that they cause a switch of the display of these references in the frame. The [input configuration \[▶ 437\]](#) offers the action "[Switch frame visualization \[▶ 414\]](#)" for this purpose. In this way it is possible to switch between several other visualization pages on a basic visualization page.

Example

A visualization page "V_MainView" has a selection menu, consisting of three buttons and a frame element. A visualization page is assigned to each button. The page is displayed in the frame element in online mode when the button is pressed.

1. Creating a selection menu with three buttons
2. Inserting a frame element
3. This frame is assigned three visualizations via the frame selection dialog, between which the display is to switch.
4. For each button an "OnClick" action of type "[Switch frame visualization \[▶ 414\]](#)" is added for the corresponding visualization page via the [input configuration \[▶ 437\]](#).



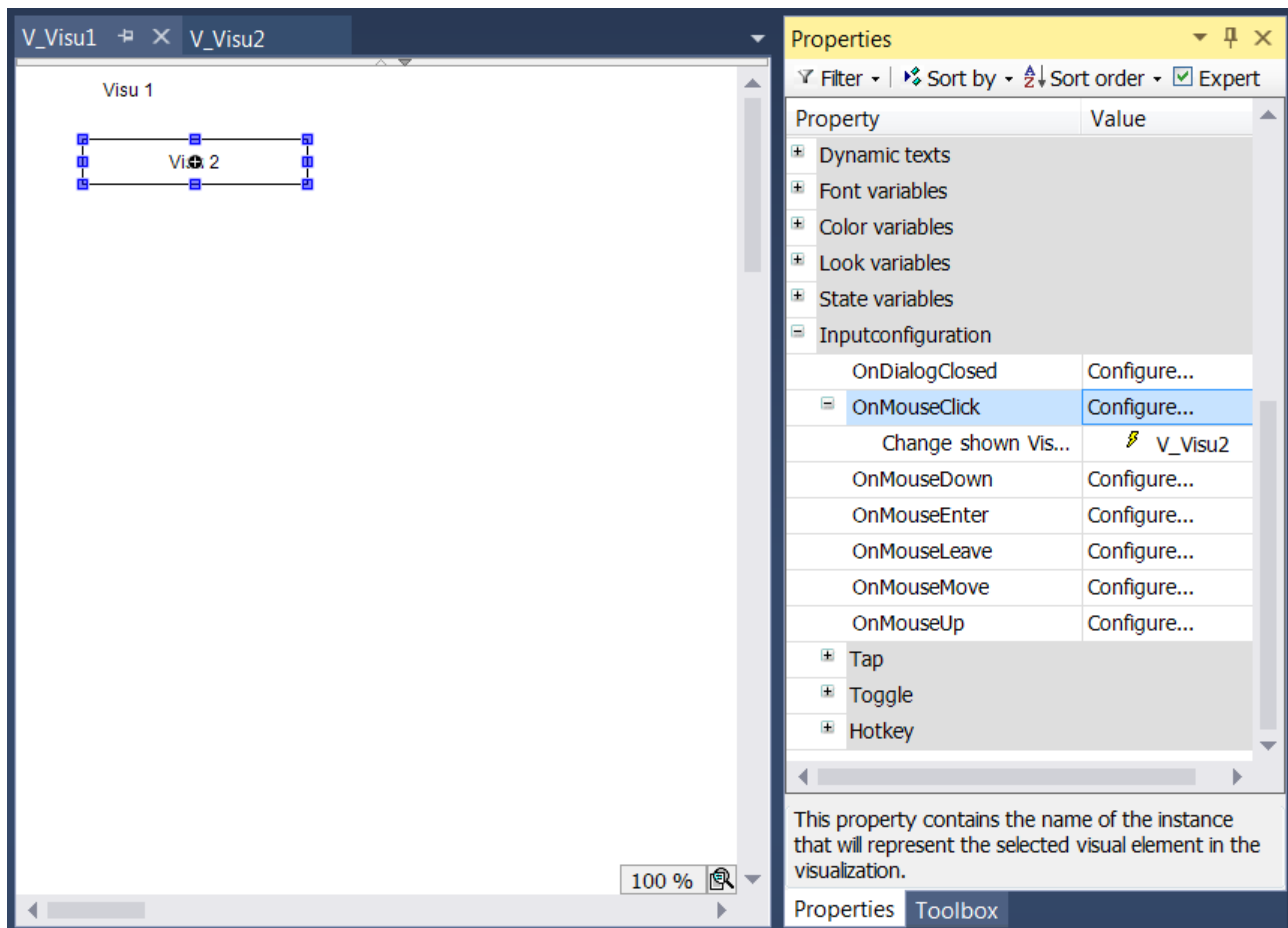
Switching between visualization pages

In addition to the switching option within a frame element, the whole currently visible visualization page can be changed. To this end, the [input configuration](#) [► 437] offers a further action with the name "[Change shown Visualization](#) [► 411]".

Example

Two visualization pages are created with the names "V_Visu1" and "V_Visu2". Each page contains a button for switching to the respective other visualization page.

1. Create two visualization objects with the names "V_Visu1" and "V_Visu2" in a PLC project.
2. Add labels on both visualization pages, so that they can be distinguished from each other.
 - "V_Visu1": Enter the text "Visu 1" in the label properties.
 - "V_Visu2": Enter the text "Visu 2" in the label properties.
3. Add a rectangle element on both sides.
 - "V_Visu1": Enter the text "Visu 2" in the rectangle properties.
 - "V_Visu2": Enter the text "Visu 1" in the rectangle properties.
4. For both rectangles configure a "[OnClick](#) [► 437]" event with the action "[Change shown Visualization](#) [► 411]".
 - "V_Visu1": Assign the visualization 'V_Visu2' to the action.
 - "V_Visu2": Assign the visualization 'V_Visu1' to the action.



CurrentVisu Variable

A page changeover is also possible via the variable [CurrentVisu](#) [► 388]. Following the assignment of a visualization page, all active clients on this page are refreshed automatically.

Assigning a variable:

```
VisuElems.CurrentVisu := sVisuName;
```

Assigning a text:

```
VisuElems.CurrentVisu := ,Visualization`;
```

15.10.2 Text and language

As a basic principle, texts and languages are managed in the PLC HMI with the aid of [text lists](#) [► 138]. ANSI and [Unicode](#) [► 388] are available as character encoding variants. In the visualization [elements](#) [► 404] can be labelled in two different ways:

- Static text
- Dynamic text

Static text

A visualization element can be assigned a static text by entering the text in the element properties under the category 'Texts'. He can also be entered directly in the Editor window, if the element is selected and the space bar is pressed. This text cannot be changed at runtime. Only changes of the language are accepted for the text.

Each static text is automatically saved in the text list "[GlobalTextList](#) [► 142]". In this text list the translations in other languages are also managed.

ID	Default	de	en
0	%s		
1	Activate	Aktivieren	Activate
2	Deactivate	Deaktivieren	Deactivate

Dynamic text

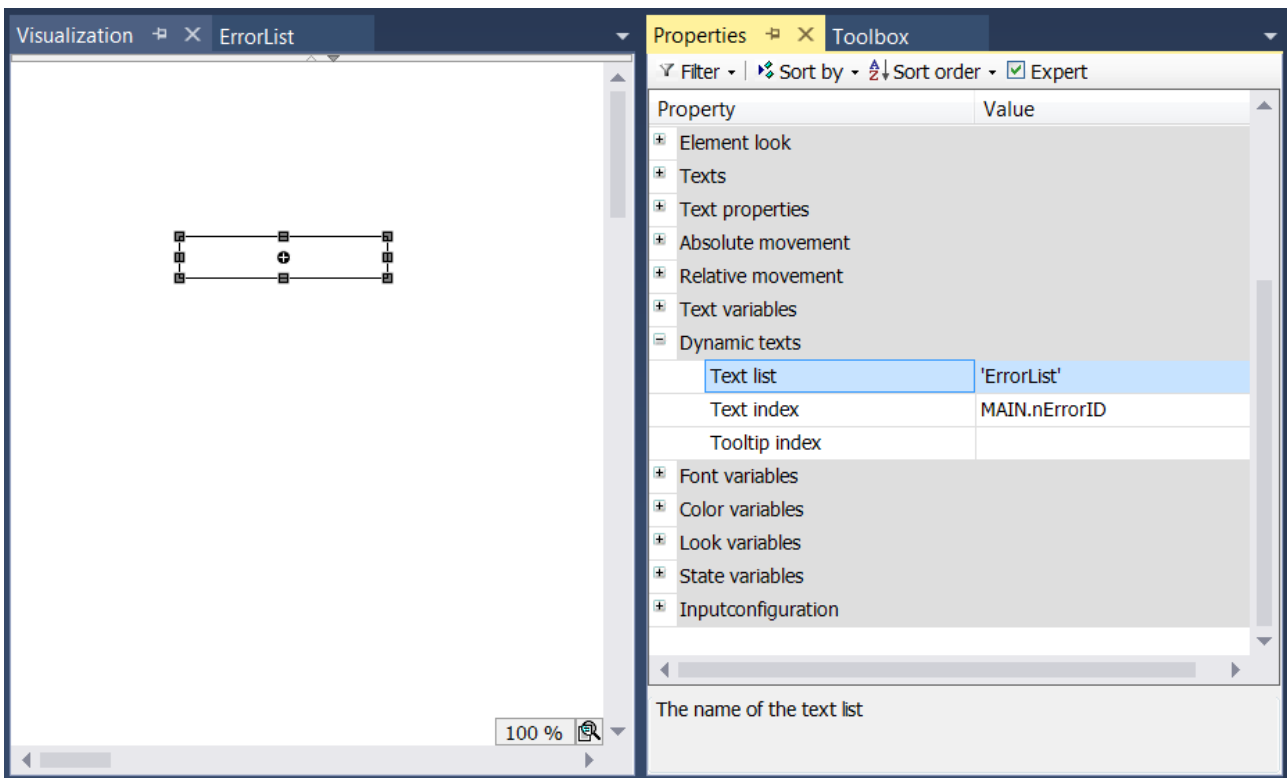
Dynamic text is entered in the element properties under the category 'Dynamic texts'. To this end the name of the text list first has to be selected from the lists that are present in the PLC project. The "GlobalTextList [▶ 142]" cannot be used here, only user-created [text lists \[▶ 138\]](#). In addition, a variable for the ID of the text entry must be specified, through which the text can be changed at runtime.

Example

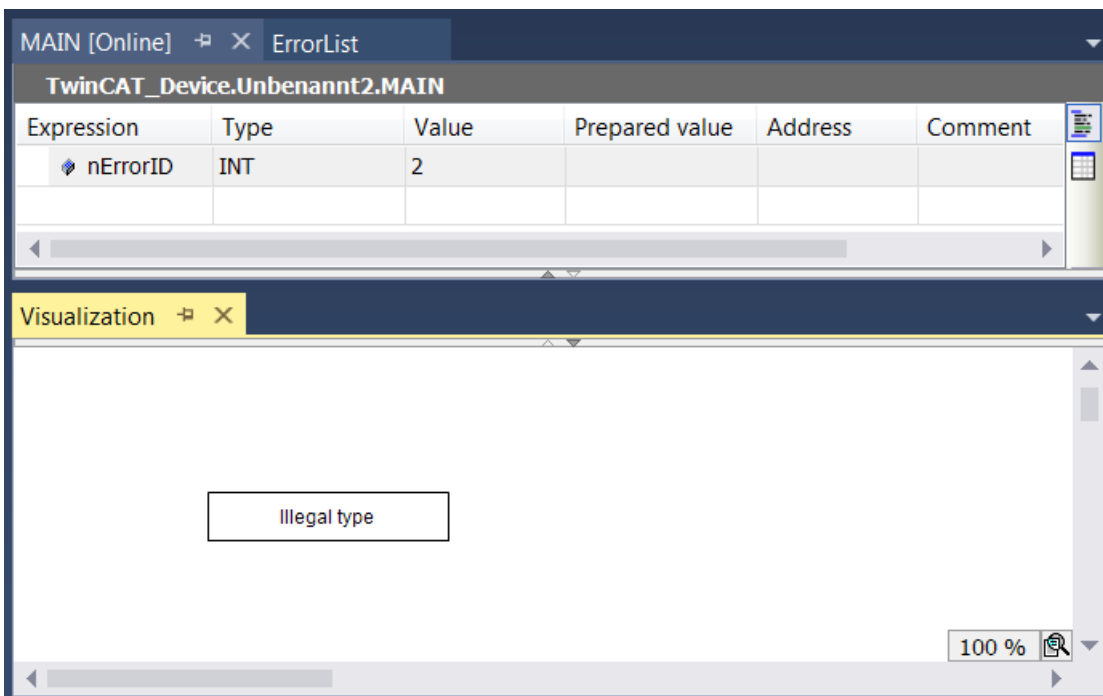
As an example, a text list with the name "ErrorList" is created. In this list a number of error messages with their translations are entered.

ID	Default	de	en
0	Wrong argument	Falsches Argument	Wrong argument
1	Bad format	Ungültiges Format	Bad format
2	Illegal type	Ungültiger Typ	Illegal type
3	Bad result	Ungültiges Ergebnis	Bad result
4	Wrong data type	Ungültiger Datentyp	Wrong data type

A [rectangle element \[▶ 475\]](#) is added to a visualization with the name "Visualization". In the properties of this element the text list "ErrorList" is selected under "[Dynamic texts \[▶ 435\]](#)" and the variable "nErrorId", which is declared in "MAIN".



At runtime one of the error messages stored in the text list "ErrorList" can be displayed in the rectangle, by setting the variable "nErrorId" to a value between or equal to 0 and less than or equal to 4. "nErrorID" has a value of 2 in the example.



Language switching

If the currently used [text list \[▶ 138\]](#) defines text translations in several languages, the [language to be used \[▶ 389\]](#) at the start of the visualization can be specified. In addition, the language can be changed at runtime with the aid of buttons on the visualization. The action "[Change the language \[▶ 411\]](#)" in the element settings under the category "Input configuration" is used for this purpose.

The variable "CURRENTLANGUAGE" from the library "[VisuElems \[▶ 401\]](#)" can be used for querying the currently used language. This variable can also be used to change the language in the program code:

```
fbTrigger(CLK := bPressed);
IF fbTrigger.Q THEN
  IF VisuElems.CURRENTLANGUAGE = 'de' THEN
    VisuElems.CURRENTLANGUAGE := 'en';
  ELSE
    VisuElems.CURRENTLANGUAGE := 'de';
  END_IF
END_IF
```



If the text list contains no entry that matches the currently set language, the entry under Default is used.



The definition of a default language at the start of the visualization and the use of the variable "VisuElems.CURRENTLANGUAGE" is only possible in conjunction with the [PLC HMI \[▶ 603\]](#) and/or the [PLC HMI Web \[▶ 608\]](#). With an [integrated visualization \[▶ 602\]](#), the text version 'Standard' is automatically used on startup. Language change at runtime via buttons on the visualization is also possible in the integrated visualization.

Formatting the text

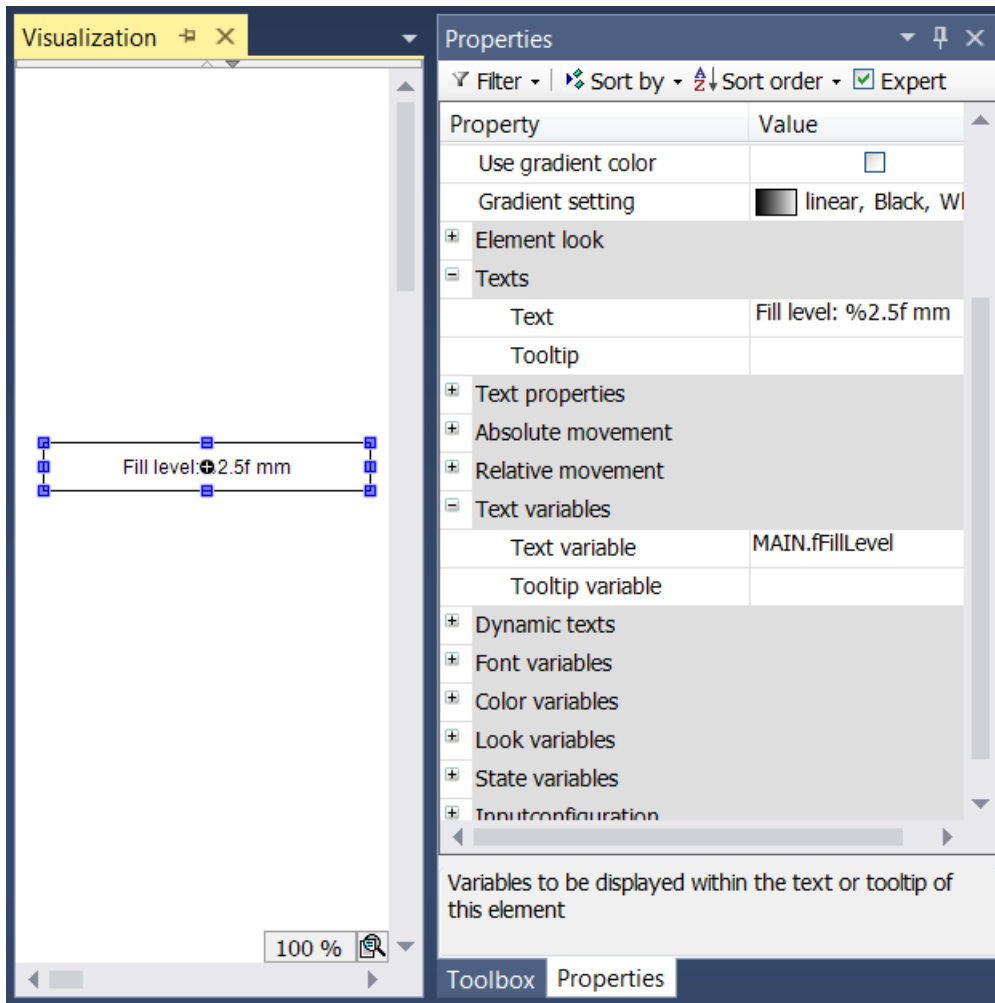
In addition to entering pure texts in the [element properties \[▶ 385\]](#) under the category 'Texts', it is also possible to add formatting information for the text display in online mode. A formatting specification always starts with "%", followed by a character that determines the formatting type. It can be used on its own or in combination with the actual text.

The variable to be output as text in the element should be specified in the element properties under the category 'Text variables'. To display the instance name of a variable, which was transferred as input parameter to a visualization function block, use the pragma attribute '[parameterstringof \[▶ 819\]](#)'.

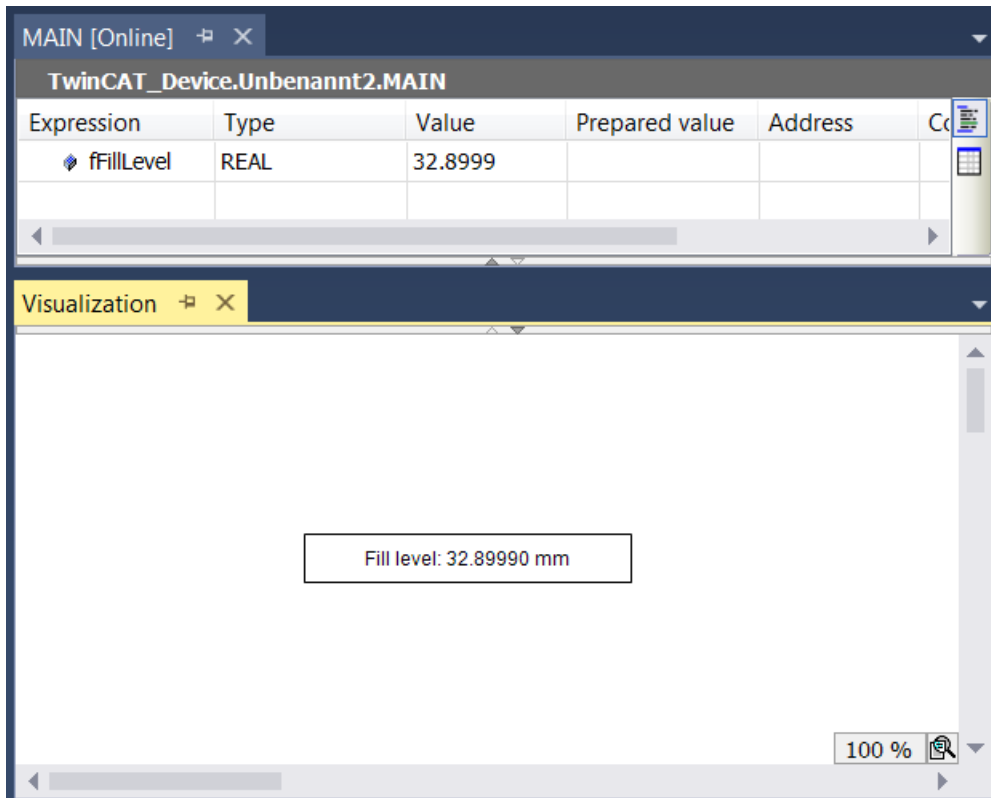
%b	Binary number
%c	Individual character
%d, %i	Decimal number
%f	REAL value
%o	Unsigned octal number (without prefixed zero)
%s	Character string
%u	Unsigned decimal number
%x	Unsigned hexadecimal number (without prefixed '0x')

Example

"Fill level: %2.5f mm" is entered in the "Text" property field in the "Texts" category of the [properties \[▶ 385\]](#) of a rectangular element. The variable to be used is specified in the input field "Text variable" under the category "Text variables". In this example it is "fFillLevel". It is declared as REAL in the program MAIN.



At runtime, the rectangle element looks as follows:





To display a percent sign % in combination with one of the formatting specifications described above, enter “%%”.

Example

Enter "Rate in %: %s" to obtain the following in online mode: "Rate in %: 12" (if the text variable currently returns "12").

Output of the system time

A combination of "%t" and the following special placeholders inside square brackets is replaced in online mode by the current system time. The placeholders define the display format.



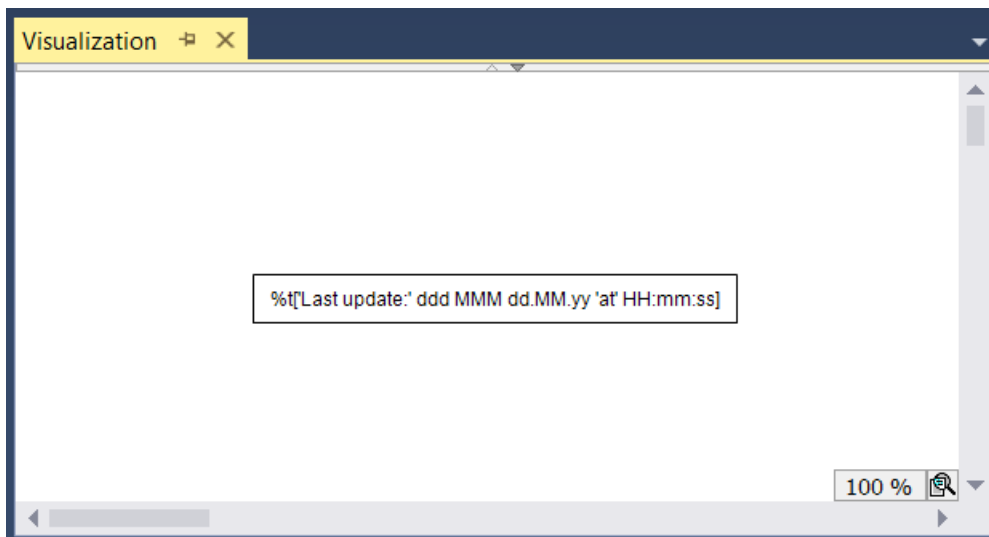
Importing TwinCAT PLC 2 Control projects: The previously used time format %t is automatically converted to the new %t[] format, if an old project is imported, although the following placeholders are no longer supported: %U, %W, %z, %Z.

Valid placeholders:

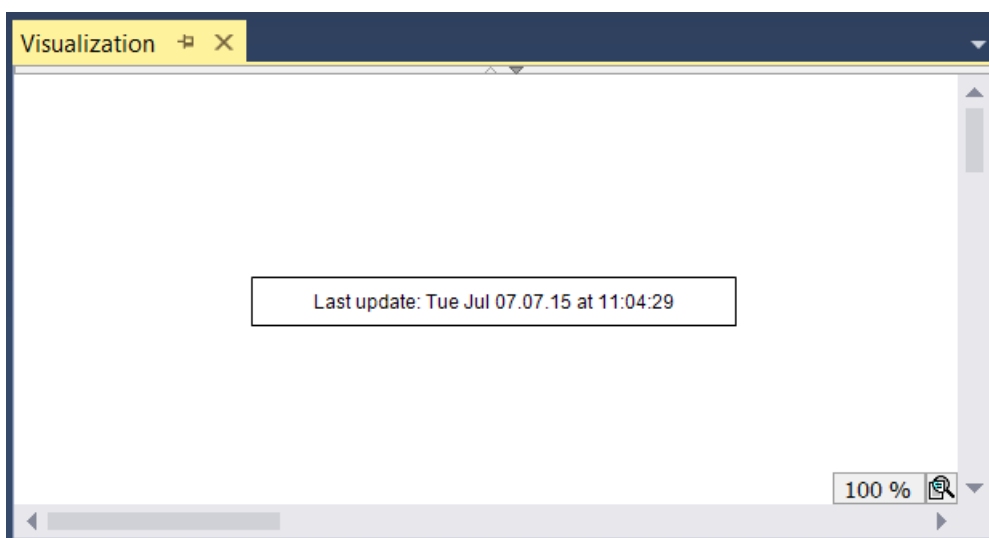
ddd	Name of the day of the week, abbreviated, e.g. "Wed"
dddd	Name of the day of the week, e.g. "Wednesday"
dddddd	Day of the week as a number (0 – 6; Sunday is 0)
MMM	Name of the month, abbreviated, e.g. "Feb"
MMMM	Name of the month, e.g. "February"
d	Day of the month as a number (1 – 31), e.g. "8"
dd	Day of the month as a number (01 – 31), e.g. "08"
M	Month as a number (1 – 12), e.g. "4"
MM	Month as a number (01 – 12), e.g. "04"
jjj	Day in the year as a number (001-366), e.g. "067"
y	Year without specification of the century (0-99), e.g. "9"
yy	Year without specification of the century (00-99), e.g. "09"
yyyy	Year with specification of the century, e.g. "2009"
HH	Hour, 24-hour format (01-24), e.g. "16"
hh	Hour, 12-hour format (01-12), e.g. "4" for 4 pm
m	Minutes (0-59), without prefixed zero, e.g. "6"
mm	Minutes (00-59), with prefixed zero, e.g. "06"
s	Seconds (0-59), without prefixed zero, e.g. "6"
ss	Seconds (00-59), with prefixed zero, e.g. "06"
ms	Milliseconds (0-999), without prefixed zero, e.g. "322"
t	ID for the display in 12-hour format: A (hours <12) or P (hours >12), e.g. "A" if the time is 9 o'clock in the morning
tt	ID for the display in 12-hour format: AM (hours <12) or PM (hours >12), e.g. "AM" if the time is 9 o'clock in the morning
' '	Strings containing one of the placeholders listed above must be enclosed in single inverted commas. All other texts within the format strings can be without inverted commas; e.g. 'update', since a "d" and a "t" are included.

Example

"%t[Last update: ddd MMM dd.MM.yy 'at' HH:mm:ss]" is entered in the property field "Text" in the category "Texts" of the [properties](#) [[▶ 385](#)] of a rectangular element.



At runtime, the rectangle element looks as follows:



Font and alignment

The font and the horizontal and vertical alignment of the element text can be defined in the [element properties](#) [▶ 385]. See categories "Font variables" and "Text properties" for the respective element.

15.10.3 Images

Images from external image files are managed in PLC projects in [image pools](#) [▶ 146]. They can basically be added to a [visualization page](#) [▶ 402] in three different ways:

- Visualization element "[Image](#) [▶ 505]"
- Visualization element "[Image switcher](#) [▶ 527]"
- [Background image](#) [▶ 378] for a visualization page

Visualization element "Image"

The element "[Image](#) [▶ 505]" enables static display of an external image file or dynamic display of different external image files on a visualization page. To this end either a [static image ID](#) [▶ 507] or a [string variable](#) [▶ 437], in which the ID of the image to be displayed at runtime is stored, can be entered in the element settings.

Visualization element "Image switcher"

The "Image switcher [▶ 527]" element can be used to create a user-defined button, which can have a button or switching function that is comparable to the standard switches [▶ 527]. Different images for the states "on", "off" and "pressed" can be defined in the element properties [▶ 528].

Background image for a visualization page

For each visualization page a background image [▶ 378] can be set. It is possible that the size of the visualization page [▶ 403] in client automatically adjusts itself based on this background image.



Folders, which provide image files for use in the visualization, are defined in the project properties [▶ 402] under the category "Visualization".

15.10.4 Keyboard operation in online mode

Some standard keyboard shortcuts are supported by every device for the keyboard operation of a visualization in online mode.

Standard keyboard shortcuts:

Key(s)	Action
[Tab]	The next element is selected according to the chronological order of insertion; within a table, each individual cell will be selected; if a frame element is selected, the selection will be then passed on to the individual elements that it contains.
[Shift + Tab]	The previous element is selected ; reverse order to [Tab]
[Enter]	Input action is performed on the selected element
[Arrow keys]	Next element is selected in the direction specified by the arrow key



For the integrated visualization, the keyboard operation can be activated and deactivated explicitly using the command "Activate keyboard operation". This may be desirable, because other commands given by keyboard shortcuts are not executed as long as the keyboard operation for the visualization is activated.



There is an option to handle keyboard events in the application code.

16 Reference Programming

16.1 Programming languages and their editors

You program a POU in the editor for the implementation language that you have selected when you created the POU. TwinCAT 3 PLC provides a text editor for ST and graphic editors for SFC, FBD/LD/IL (textual) and CFC.

Open the editor by double-clicking on the POU in the PLC project tree or with the command **Open** in the context menu.

Each of the programming language editors consists of two subwindows:



- The upper part is used for the declarations in the declaration editor, either in text-based or tabular form, depending on the setting.
- In the lower part you insert the implementation code in the respective language.

The display and the behavior of each editor can be configured for the entire project in the corresponding tab of the TwinCAT options.

16.1.1 Declaration Editor

In the declaration editor, you declare variables in variable lists and POU's.

If the declaration editor is used in conjunction with a programming language editor, it opens in a window above the programming language editor.

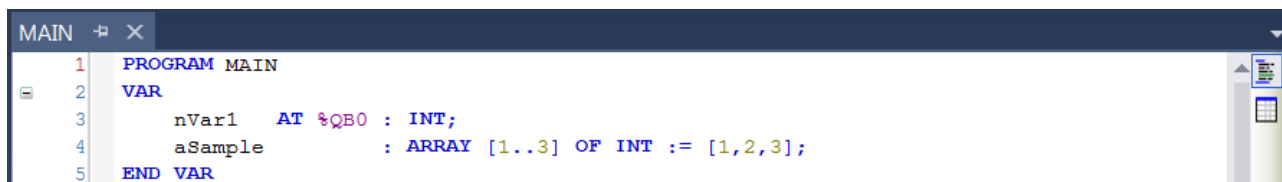
The declaration editor offers two possible views: textual () and tabular (). In the dialog **Tools > Options > TwinCAT > PLC Environment > Declaration editor** you can define whether either only the text-based view or only the tabular view is available, or whether the user can choose between the two views via the buttons on the right of the editor window.

Rectangle selection is possible in the textual view of the declaration editor. The shortcuts for the rectangle selection can be found in the chapter [ST Editor](#) [▶ 624].

Text-based declaration editor

You configure the behavior and appearance of the text editor with the settings in the dialog **Tools > Options > TwinCAT > PLC Environment > Text editor**. The settings concern colors, line numbers, tab widths, indentations, etc. The normal Windows functions and, if applicable, the IntelliMouse functions are available in the text editor.

You can also use comments in the textual declaration editor (see [ST comments](#) [▶ 634]).

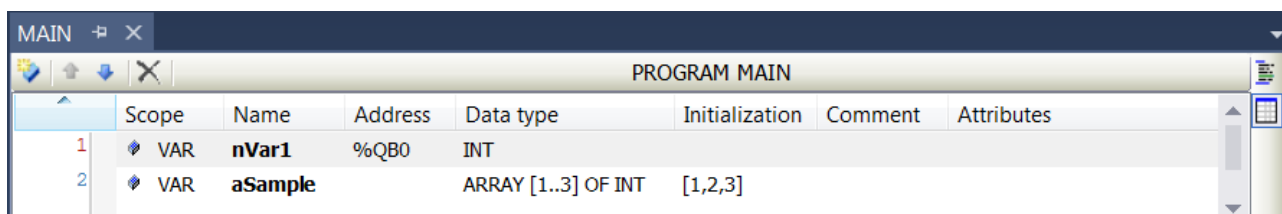


```

MAIN
1 PROGRAM MAIN
2 VAR
3     nVar1 AT %QB0 : INT;
4     aSample : ARRAY [1..3] OF INT := [1,2,3];
5 END_VAR
  
```

Tabular declaration editor



In the tabular declaration editor, you insert variable declarations in a table with the following columns: scope, name, address, data type, initialization, comment and (pragma) attributes.



	Scope	Name	Address	Data type	Initialization	Comment	Attributes
1	VAR	nVar1	%QB0	INT			
2	VAR	aSample		ARRAY [1..3] OF INT	[1,2,3]		

Declaration editor in online mode

In online mode, you see the tabular view of the editor. The header always contains the current object path: <device name>.<project name>.<object name>. As opposed to offline mode, the table also contains the columns **Value** and **Prepared value**.






Value	<p>Shows the actual value on the controller, thus provides monitoring functionality.</p> <p>If the expression is an array with more than 1000 elements, you can specify the range of the array indices to be monitored. To do this, double-click in the column Data type to open the dialog Monitoring area [▶ 224]. In this dialog, the declared array range is entered as the Valid range for monitoring. A maximum of 20000 elements can be displayed per array.</p> <p>You define the range of array indices to be displayed by specifying the start and end index. To move this area more easily while keeping the scope constant, you can use the scrollbars coupled for this purpose. To switch between linked scrollbars  and unlinked scrollbars  click the icon to the right of the bars. In the uncoupled state, you can increase or decrease the scope of the area to be displayed as you wish.</p>
Prepared value	<p>Contains the value you may have prepared for forcing or writing [▶ 214].</p> <p>If you double-click a field in the column Prepared value, you can directly enter a value for writing or forcing.</p> <p>For enumerations a combo box opens, from which you can select a value.</p> <p>With a boolean variable you can toggle (switchover) the prepared value with the help of the [Enter] or [Space] key.</p> <p>If an expression (variable) is of a structured data type, for example the instance of a function block or an array variable, then it is preceded by a plus or minus sign.</p> <p>You can adjust the format of the representation of floating-point numbers as well as the representation of the inheritance hierarchy in the context menu.</p>

See also:

- [Declaring variables](#) [▶ 66]
- [Forcing in the declaration part](#) [▶ 217]
- Documentation TC3 User Interface: [Options dialog - declaration editor](#) [▶ 969]
- Documentation TC3 User Interface: [Prepare dialog value](#) [▶ 961]

16.1.2 Common Functions in Graphical Editors

The implementation part of the graphical editors for FBD, LD, CFC and SFC has a toolbar in the lower right corner:

	Back to normal editing mode: The mouse pointer changes to the default arrow shape. You can again select and edit elements in the editor window.
	Panning tool: The mouse pointer changes to the shape of two crossed arrows. You can now click anywhere in the editor window, keep the mouse button pressed then move the visible area of the FBD/LD/IL editor or the CFC diagram within the window.
	Magnifier tool: An additional magnifier window opens in the lower right corner of the editor window and the mouse pointer changes to the shape of two crossed arrows. As long as you now move the mouse pointer over your diagram, the magnifier shows the area of the diagram around the pointer at 100% size. If you click into the window, the magnifier closes, and the part of the diagram that was shown in it is displayed at 100% size. Therefore, if you want to retain the set zoom factor, you should use  to return to normal editing mode.
	Zoom tool: The button opens a submenu for selecting a zoom factor. Selecting the three points ... opens the zoom dialog, where you can enter a different value. The current zoom factor is always displayed to the left of the button.

Zooming in/out with the mouse wheel: If you press the **[Ctrl]** button while you move the mouse wheel, the zoom factor changes in 10% steps.

Each graphical editor features a toolbox (**Toolbox** view), by default to the right of the editor window. The toolbox contains the available elements that you can drag with the mouse to the respective insertion positions in the editor window. TwinCAT highlights the respective insert positions with gray diamond, triangular or arrow-shaped position marks. As soon as the mouse pointer is over one of these marks, the mark turns green. When you now release the mouse button, TwinCAT inserts the element at the current position.

Moving elements in the editor is also possible with the mouse.

You can drag declarations of function blocks into the editor window in the FBD, LD and CFC graphical editors. To do this, select the complete declaration (variable name and data type) and drag it to a suitable place in the editor window. With the Ladder Diagram, you can additionally drag boolean declarations into the editor and insert them as a contact.

See also:

- [SFC editor \[▶ 635\]](#)
- [FBD/LD/IL editor \[▶ 651\]](#)
- [CFC editor \[▶ 666\]](#)

16.1.3 Structured Text and Extended Structured Text (ExST)

16.1.3.1 ST Editor

The ST Editor is a text editor and is used to implement code in Structured Text (ST) and Extended Structured Text (ExST).

The line numbering is located on the left edge of the editor. When entering the programming elements, the function **List components** and the input assistant (**F2**) will help you. If the cursor is on a variable, TwinCAT displays information on the variable declaration in a tooltip.

You can perform a rectangle selection with the following shortcuts:

- **[Shift] + [Alt] + [Right arrow]**: The selected area will be extended one place to the right.
- **[Shift] + [Alt] + [Left arrow]**: The selected area will be extended one place to the left.
- **[Shift] + [Alt] + [Up arrow]**: The selected area will be extended one line upwards.
- **[Shift] + [Alt] + [Down arrow]**: The selected area will be extended one place downwards.

Zooming in the editor window is possible by turning the mouse wheel while holding down the **Ctrl** key.

You can configure the behavior (such as parentheses, mouse actions, tabulators) and appearance of the editor in the TwinCAT options in the category **PLC Environment > Text editor**.

For an incremental search for strings within the editor, use the shortcut **[Ctrl] + [Shift] + [I]** to open an input field at the bottom of the editor. As soon as you start typing a string, the editor highlights the corresponding occurrences in color. To the right of the input field is the number of matches found. Using the arrow buttons or the shortcuts **[Alt] + [Page up]** or **[Alt] + [Page down]** you can place the cursor on a found location.

When you place the cursor on a symbol name, all the places where the symbol is used within the editor are highlighted. The references correspond to the hits in the cross-reference list. For very large projects, this can lead to delays in the input. In this case you can disable the function in the options of the text editor.

TwinCAT recognizes syntax errors during the editing process and underlines the errors with a red squiggly line. Prerequisite is that the corresponding option is activated in the TwinCAT options, category **PLC Environment > Smart Coding**. Any syntax errors detected during the compilation are displayed in the **Error list (View menu)**.

See also:

- [Programming Structured Text \(ST\) \[► 124\]](#)
- [ST Expressions \[► 625\]](#)
- [Assignments \[► 627\]](#)
- [Instructions \[► 630\]](#)
- [TC3 User Interface documentation: Dialog Options - Text editor \[► 985\]](#)

16.1.3.2 ST editor in online mode

In online mode, TwinCAT displays the variables and expressions used in the ST editor. Writing and forcing of variables as well as expressions and debugging functions (breakpoints, single-step processing) are also possible.

If you use assignments as expressions in ST programming, no further breakpoint positions occur within a line.

See also:

- [Monitoring in Programming Objects \[► 222\]](#)
- [Use of breakpoints \[► 211\]](#)
- [Using Watchlists \[► 226\]](#)
- [Forcing and Writing Variables Values \[► 214\]](#)
- [Testing a PLC project and troubleshooting \[► 211\]](#)
- [Stepwise processing of the program \(stepping\) \[► 213\]](#)
- [Flow Control \[► 219\]](#)
- [Determining the Current Processing Position with the Call Stack \[► 221\]](#)

16.1.3.3 ST Expressions

An expression is a construct that returns a value after it was evaluated.

Expressions are composed of operators and operands. An operand can be a constant, a variable, a function call or another expression.

Examples:

2014	(* constant *)
nVar	(* variable *)
F_Fct (a, b)	(* function call *)
(x*y)/z	(* expression *)

See also:

- [Operators \[▶ 696\]](#)
- [Operands \[▶ 744\]](#)

Evaluation of Expressions

The evaluation of an expression is carried out by processing the operators according to certain binding rules. TwinCAT first processes the operator with the strongest binding. Operators with the same binding strength are processed from left to right.

Operation	Symbol	Binding strength
Put in parentheses	(Expression)	Strongest binding
Function call	Function name (parameter list) all operators with syntax: <operator> ()	
Exponentiation	EXPT	
Negate	-	
Build. complements	NOT	
Multiply	*	
Divide	/	
Modulo	MOD	
Add	+	
Subtract	-	
Compare	<, >, <=, >=	
Equal to	=	
Not Equal to	<>	
Bool AND	AND AND_THEN	
Bool XOR	XOR	Weakest binding
Bool OR	OR OR_ELSE	

Sample:

In the following sample the operators [AND_THEN \[▶ 713\]](#) and [OR \[▶ 713\]](#) are used. Note that OR has the weakest binding and that TwinCAT executes expressions at other operands of the AND_THEN operator only if the first operand of the AND_THEN operator is TRUE.

Therefore, the results for the four expressions shown here are as follows:

- The pointer "pSample" is dereferenced for none of the four expressions. The background is that the AND_THEN operator is used, and dereferencing would take place at the other operands of the AND_THEN operator. However, since the first operand of the AND_THEN operator already returns FALSE (since "pSample" has the value 0), the further AND_THEN operands and thus the pointer dereferences are not executed. Using the AND_THEN operator avoids a null pointer exception at runtime. If the AND operator was used instead of the AND_THEN operator, the pointer "pSample" would be dereferenced as a null pointer within the operands, which would result in a null pointer exception.
- In expressions 1 and 2, OR follows as a further operator after the AND_THEN operations. Therefore, the counters "nCounter1" and "nCounter2" increment if "pSample" is 0 and "bTest" is TRUE. The abbreviation of the expressions is "IF <FALSE> OR TRUE THEN", which returns TRUE.
- In expressions 3 and 4, on the other hand, "nCounter3" and "nCounter4" do not increment if "pSample" is 0 and "bTest" is TRUE, since the first operand, "pSample <> 0", returns FALSE. Due to AND_THEN and the parentheses, no further operands or operators are checked. The abbreviated form of the expressions is "IF <FALSE> AND_THEN <...>", which returns FALSE.

```

PROGRAM MAIN
VAR
    pSample      : POINTER TO INT;
    bTest        : BOOL := TRUE;
    nCounter1    : INT;
    nCounter2    : INT;
    nCounter3    : INT;
    nCounter4    : INT;
END_VAR

// Expression 1
IF (pSample <> 0) AND_THEN (pSample^ = 250) AND_THEN (pSample^ <> 300) OR bTest THEN
    nCounter1 := nCounter1 + 1; // increasing if (pSample = 0) and (bTest = TRUE)
END_IF

// Expression 2
IF ((pSample <> 0) AND_THEN (pSample^ = 250) AND_THEN (pSample^ <> 300)) OR bTest THEN
    nCounter2 := nCounter2 + 1; // increasing if (pSample = 0) and (bTest = TRUE)
END_IF

// Expression 3
IF (pSample <> 0) AND_THEN ((pSample^ = 250) AND_THEN (pSample^ <> 300) OR bTest) THEN
    nCounter3 := nCounter3 + 1; // not increasing if (pSample = 0) and (bTest = TRUE)
END_IF

// Expression 4
IF (pSample <> 0) AND_THEN ((pSample^ = 250) OR bTest) THEN
    nCounter4 := nCounter4 + 1; // not increasing if (pSample = 0) and (bTest = TRUE)
END_IF

```

16.1.3.4 Assignments

16.1.3.4.1 ST assignment operator

Syntax:

<operand> := <expression>

This assignment operator performs the same function as the MOVE operator.

See also:

- [MOVE \[▶ 706\]](#)

16.1.3.4.2 ST assignment operator for outputs

The assignment operator => assigns the output of a function, a function block, or a method to a variable. The space to the right of the operator can be empty.

Syntax:

<output> => <variable>

Sample:

```

bFBCompOutput1 => bVar1;
bFBCompOutput2 => ;

```

bFBCompOutput1 and bFBCompOutput2 are outputs of a function block. The value of bFBCompOutput1 is assigned to the variable bVar1.

16.1.3.4.3 ExST assignment S=

If the operand of the Set assignment switches to TRUE, the assignment causes the variable to the left of the operator to be assigned a TRUE. The variable is set.

Syntax:

<variable name> S= <operand name> ;

The variable and the operand have the data type BOOL.

Sample:

```
PROGRAM MAIN
VAR
  bOperand      : BOOL := FALSE;
  bSetVariable  : BOOL := FALSE;
END_VAR
bSetVariable S= bOperand;
```

If the operand bOperand switches from FALSE to TRUE, the variable bSetVariable is assigned a TRUE. But then the variable keeps this state even if the operand continues to change its state.

Multiple assignments

In the case of multiple assignments within a code line, the individual assignments are not processed from right to left, but all assignments refer to the operand at the end of the code line.

Sample:

```
FUNCTION F_Sample: BOOL
VAR_INPUT
  bIn : BOOL;
END_VAR
IF bIn = TRUE THEN
  F_Sample := TRUE;
  RETURN;
END_IF
PROGRAM MAIN
VAR
  bSetVariable   : BOOL;
  bResetVariable : BOOL := TRUE;
  bVar           : BOOL;
END_VAR
bSetVariable S= bResetVariable R= F_Sample(bIn := bVar);
```

bResetVariable receives the R= assignment of the return value of F_Sample. bSetVariable receives the S= assignment of the return value of F_Sample but not of bResetVariable.

16.1.3.4.4 ExST assignment R=

If the operand of the reset assignment switches to TRUE, the assignment causes the variable to the left of the operator to be assigned a FALSE. The variable is reset.

Syntax:

<variable name> R= <operand name> ;

The variable and the operand have the data type BOOL.

Sample:

```
PROGRAM MAIN
VAR
  bOperand      : BOOL := FALSE;
  bResetVariable : BOOL := TRUE;
END_VAR
bResetVariable R= bOperand;
```

If the operand bOperand switches from FALSE to TRUE, the variable bResetVariable is assigned a FALSE. But then the variable keeps its state even if the operand keeps changing its state.

Multiple assignments

In the case of multiple assignments within a code line, the individual assignments are not processed from right to left, but all assignments refer to the operand at the end of the code line.

Sample:


```

FUNCTION F_Sample: BOOL
VAR_INPUT
    bIn : BOOL;
END_VAR

IF bIn = TRUE THEN
    F_Sample := TRUE;
    RETURN;
END_IF

PROGRAM MAIN
VAR
    bSetVariable : BOOL;
    bResetVariable : BOOL := TRUE;
    bVar : BOOL;
END_VAR

bSetVariable S= bResetVariable R= F_Sample(bIn := bVar);
    
```

bResetVariable receives the R= assignment of the return value of F_Sample. bSetVariable receives the S= assignment of the return value of F_Sample but not of bResetVariable.

16.1.3.4.5 ExST assignment as expression

In ExST, TwinCAT allows the use of assignments as expressions, in extension to the IEC 61131-3 standard.

Examples:

nVarInt1 := nVarInt2 := nVarInt3 + 9;	(*nVarInt1 and nVarInt2 obtain the value of nVarInt3 + 9*)
fVarReal1 := fVarReal2 := nVarInt;	(*fVarReal1 and fVarReal2 obtain the value of nVarInt*)
nVarInt:= fVarReal1:= nVarInt;	(*incorrect assignment, the data types do not match!*)
IF b := (i = 1) THEN i := i + 1; END_IF	(*b obtains the value of the Boolean expression i = 1 and is then checked in the if query*)

16.1.3.4.6 Assignment operator REF=

The operator generates a [reference \[► 770\]](#) (pointer) to a value.

Syntax:

```
<variable name> REF= <variable name>
```

Sample:

```

PROGRAM MAIN
VAR
    refA : REFERENCE TO ST_Sample;
    stA : ST_Sample;
    refB : REFERENCE TO ST_Sample;
    stB1 : ST_Sample;
    stB2 : ST_Sample;
END_VAR

refA REF= stA; // represents => refA := ADR(stA);
refB REF= stB1; // represents => refB := ADR(stB1);
refA := refB; // represents => refA^ := refB^; (value assignment of refB as refA and refB are implicitly dereferenced)
refB := stB2; // represents => refB^ := stB2; (value assignment of stB2 as refB is implicitly dereferenced)
END_VAR
    
```

See also:

- [REFERENCE \[► 770\]](#)

16.1.3.5 Instructions

16.1.3.5.1 ST instruction IF

The IF statement is used to test a condition and to execute the subsequent instructions if the condition is met.

A condition is encoded as a [expression \[► 625\]](#) that returns a boolean value. If the expression returns TRUE, the condition is met and the associated statements after THEN are executed. If the expression returns FALSE, the following conditions marked ELSIF are evaluated. If an ELSIF condition returns TRUE, the instructions after the associated THEN are executed. If all conditions are FALSE, the statements after ELSE are executed.

So at most one branch of the IF statement is executed. The ELSIF branches and the ELSE branch are optional.

Syntax

```
IF <condition> THEN
    <statements>
(ELSIF <condition> THEN
    <statements>)*
(ELSE
    <statements>)?
END_IF;

// ( ... ) * None, once or several times
// ( ... ) ? Optional
```

Sample:

```
PROGRAM MAIN
VAR
    nTemp      : INT;
    bHeatingOn : BOOL;
    bOpenWindow : BOOL;
END_VAR

IF nTemp < 17 THEN
    bHeatingOn := TRUE;
ELSIF nTemp > 25 THEN
    bOpenWindow := TRUE;
ELSE
    bHeatingOn := FALSE;
    bOpenWindow := FALSE;
END_IF;
```

At runtime, the program is run through as follows:

If the evaluation of the expression `nTemp < 17 = TRUE`, the following instruction is executed and the heating is switched on. If the evaluation of the expression `nTemp < 17 = FALSE`, the following ELSIF condition `nTemp > 25` is evaluated. If this is true, the statement under ELSIF is executed and the window is opened. If all conditions are FALSE, the statements under ELSE are executed. The heating is turned off and the window is closed.

See also:

- [ST expressions \[► 625\]](#)

16.1.3.5.2 ST instruction FOR

The FOR loop is used to execute instructions with a specific number of retries.

Syntax:

```
FOR <counter> := <start value> TO <end value> {BY <increment> } DO
  <instructions>
END_FOR;
```

The section inside the curly brackets {} is optional.

TwinCAT executes the <instructions> until the <counter> is not greater, or - if the step size increment is negative - is less than the <end value>. This is checked before the <instructions> are executed.

Whenever the <instructions> have been executed, the <counter> is automatically incremented by the step size <increment>. The step size <increment> can have any integer value. If you do not specify a step size, the default step size 1 is used.

Example:

```
FOR nCounter := 1 TO 5 BY 1 DO
  nVar1 := nVar1*2;
END_FOR;
nErg := nVar1;
```

If you have set nVar1 to 1, nVar1 has the value 32 after the FOR loop.



End value of the FOR loop

The <end value> must not have the same value as the upper bound of the data type of the counter.

In addition to the IEC 61131-3 standard, you can use the CONTINUE instruction within the FOR loop.

See also:

- [ExST instruction CONTINUE \[► 634\]](#)
- [Integer Data Types \[► 758\]](#)

16.1.3.5.3 ST instruction CASE

The CASE instruction is used to group multiple conditional instructions with the same conditional variable in a construct.

Syntax:

```
CASE <Var1> OF
<value1>:<instruction1>
<value2>:<instruction2>
<value3, value4, value5>:<instruction3>
<value6 ... value10>:<instruction4>
...
<value n>:<instruction n>
{ELSE <ELSE-instruction>}
END_CASE;
```

The section inside the curly braces {} is optional.

Processing scheme of a CASE instruction:

- If the variable <Var1> has the value <value i>, <instruction i> is executed.
- If the variable <Var1> has none of the specified values, the <ELSE instruction> is executed.
- If you want to execute the same instruction for multiple values of the variable, you can write these values separated by commas.

Example:

```
CASE nVar OF
  1,5 : bVar1 := TRUE;
      bVar3 := FALSE;

  2 : bVar2 := FALSE;
      bVar3 := TRUE;
```

```

10..20 : bVar1 := TRUE;
        bVar3 = TRUE;
ELSE
        bVar1 := NOT bVar1;
        bVar2 := bVar1 OR bVar2;
END_CASE;

```

16.1.3.5.4 ST instruction WHILE

Like the FOR loop, the WHILE loop is used to execute instructions repeatedly until the termination condition applies. The termination condition of a WHILE loop is a Boolean expression.

Syntax:

```

WHILE <boolean expression> DO
  <instructions>
END_WHILE;

```

TwinCAT executes the <instructions> repeatedly as long as the Boolean expression <boolean expression> returns TRUE. If the Boolean expression is FALSE at the first evaluation, then TwinCAT never executes the instructions. If the Boolean expression never takes the value FALSE, the instructions are repeated continuously, causing a runtime error.

Example:

```

WHILE nCounter <> 0 DO
  nVar1 := nVar1*2
  nCounter := nCounter-1;
END_WHILE;

```



You must ensure programmatically that no infinite loop is created.

The WHILE and REPEAT loops are, in a sense, more powerful than the FOR loop, since you do not need to know the number of iterations before you run the loop. In some cases, it is thus only possible to work with these two types of loops. However, if the number of iterations is known, a FOR loop is preferable to avoid infinite loops.

In addition to the IEC 61131-3 standard, you can use the CONTINUE instruction within the WHILE loop.

See also:

- [ExST instruction CONTINUE \[► 634\]](#)
- [ST instruction FOR \[► 630\]](#)

16.1.3.5.5 ST instruction REPEAT

You use the REPEAT loop in the same way as the WHILE loop, but with the difference that TwinCAT does not check the termination condition until after the loop has been executed. This behavior has the consequence that the REPEAT loop will run at least once, no matter what the terminating condition is.

Syntax:

```

REPEAT
  <instructions>
UNTIL <boolean expression>
END_REPEAT;

```

TwinCAT executes the <instructions> repeatedly until the <boolean expression> returns TRUE.

If the Boolean expression is TRUE at the first evaluation, then TwinCAT executes the instructions once. If the Boolean expression never takes the value TRUE, the instructions are repeated continuously, causing a runtime error.

Example:

```
REPEAT
  nVar1 := nVar1*2;
  nCounter := nCounter-1;
UNTIL
  nCounter = 0
END_REPEAT;
```

The WHILE and REPEAT loops are, in a sense, more powerful than the FOR loop, since you do not need to know the number of iterations before you run the loop. In some cases, you can work only with these two types of loops. However, if the number of iterations is known, a FOR loop is preferable to avoid infinite loops.

In addition to the IEC 61131-3 standard, you can use the CONTINUE instruction within the WHILE loop.

See also:

- [ExST instruction CONTINUE \[► 634\]](#)
- [ST instruction FOR \[► 630\]](#)
- [ST instruction WHILE \[► 632\]](#)

16.1.3.5.6 ST instruction RETURN

You use the RETURN instruction to exit a function block. You can make this dependent on a condition, for example.

Sample:

```
IF bVar1 = TRUE THEN
  RETURN;
END_IF;
nVar2 := nVar2 + 1;
```

If the value of bVar1 is TRUE, the function block is exited immediately and TwinCAT does not execute the instruction `nVar2 := nVar2 + 1;`.

See also:

- [ST instruction IF \[► 630\]](#)

16.1.3.5.7 ST instruction JMP

The JMP instruction is used to perform an unconditional jump to a line, which is marked by a label.

Syntax:

```
<label>: <instructions>
JMP <label>;
```

The <label> is a freely selectable, unique identifier, which you place at the beginning of a line. Reaching the JMP instruction triggers a jump back to the line with the <label>.

Example:

```
nVar1 := 0;
_label1 : nVar1 := nVar1+1;
(*instructions*)
IF (nVar1 < 10) THEN
  JMP _label1;
END_IF;
```



You must ensure programmatically that no infinite loop is created. For example, you can link the jump to a condition.

16.1.3.5.8 ST instruction EXIT

Use the EXIT instruction in a FOR, WHILE or REPEAT loop to exit the loop, notwithstanding other termination conditions.

See also:

- [ST instruction FOR \[► 630\]](#)
- [ST instruction REPEAT \[► 632\]](#)
- [ST instruction WHILE \[► 632\]](#)

16.1.3.5.9 ExST instruction CONTINUE

CONTINUE is an Extended Structured Text (ExST) instruction.

Use the instruction in FOR, WHILE and REPEAT loops to trigger a jump to the start of the next loop.

Sample:

```
FOR nCounter :=1 TO 5 BY 1 DO
  nInt1:=nInt1/2;
  IF nInt1=0 THEN
    CONTINUE; (* to avoid a division by zero *)
  END_IF
  nVar1:=nVar1/nInt1; (* executed, if nInt1 is not 0 *)
END_FOR;
nRes:=nVar1;
```

See also:

- [ST instruction FOR \[► 630\]](#)
- [ST instruction WHILE \[► 632\]](#)
- [ST instruction REPEAT \[► 632\]](#)

16.1.3.5.10 ST call function block**Syntax:**

<FB-instance>(<FB input variable>:=<value or adress>|, <other FB input variables>);

Example:

```
fbTMR : TON;
fbTMR (IN := %OX5, PT := T#300ms);
bVarA := fbTMR.Q;
```

The timer function block TON is instantiated in `fbTMR : TON;` and called with assignments for the parameters IN and PT.

Output Q is addressed with `fbTMR.Q` and assigned to the variable `bVarA`.

See also:

- [Object Function block \[► 84\]](#)

16.1.3.5.11 ST comments

Comment	Description	Sample:
Single line	Starts with // and ends at the end of the line.	// Dies ist ein Kommentar
Multi-line	Starts with (* and ends with *).	(* Dies ist ein mehrzeiliger Kommentar *)
Nested	Starts with (* and ends with *). Additional (*... *) comments can be contained within this comment.	(* a:=fbTest.out; (* 1.Kommentar *) b:=b+1; (* 2.Kommentar *)*)

In addition, one or more marked lines can be commented in/out with a menu command or a shortcut:

- [Command Comment Selection \[► 882\]](#)
- [Command Uncomment Selection \[► 882\]](#)

Tooltips comments

You can create a tooltip for a POU by inserting a comment above the declaration of the POU.

Sample:

```
// Das ist ein Beispiel-Funktionsbaustein
FUNCTION_BLOCK FB_Sample
VAR_INPUT
END_VAR
```

16.1.4 Sequential Function Chart (SFC)

16.1.4.1 SFC Editor

The SFC editor is a graphic editor. A newly added SFC POU contains an init step and a subsequent transition.

In the SFC editor, you can insert the individual elements into a diagram using commands from the **SFC** menu, the context menu or the **Toolbox** view.

When inserting using a menu command, the elements that can be inserted at the currently selected position are available for selection.

Before inserting branches in parallel to several actions and transitions, you need to select these actions and transitions through multiple selection.

You can also drag and drop SFC elements from the **Toolbox** view into the diagram. As soon as you drag an element into the editor, TwinCAT marks all possible insert positions with gray rectangles. If you move the mouse over a possible insert position, the color of the rectangle changes to green. If you release the mouse button, the element will be inserted at this point.

If you insert a branch using drag-and-drop, you must mark the beginning and end of the branch with the mouse. You mark the beginning of the branch by releasing the mouse at an insert position. The color of the rectangle changes to red. The end of the branch is defined by clicking on a second insert position. TwinCAT then inserts a branch around the objects between the start mark and end mark.

For copying step and transition elements, which call action objects or transition objects, you can set two different duplication modes. The references are either copied, or the referenced objects are "embedded" and duplicated during copying.

The behavior and appearance of the editor can be defined in the TwinCAT options in the category **TwinCAT > PLC Environment > SFC editor**.

See also:

- [Common Functions in Graphical Editors \[▶ 623\]](#)
- [Programming in Sequential Function Chart \(SFC\) \[▶ 124\]](#)
- TC3 User Interface documentation: [SFC \[▶ 1001\]](#)
- TC3 User Interface documentation: [Dialog Options - SFC editor \[▶ 977\]](#)

16.1.4.2 SFC editor in online mode

In the SFC editor, you can display the variables and expressions used at runtime on the controller. You can also write and force the variables and expressions. Debugging functionality (breakpoints, step-by-step execution etc.) is not yet available.

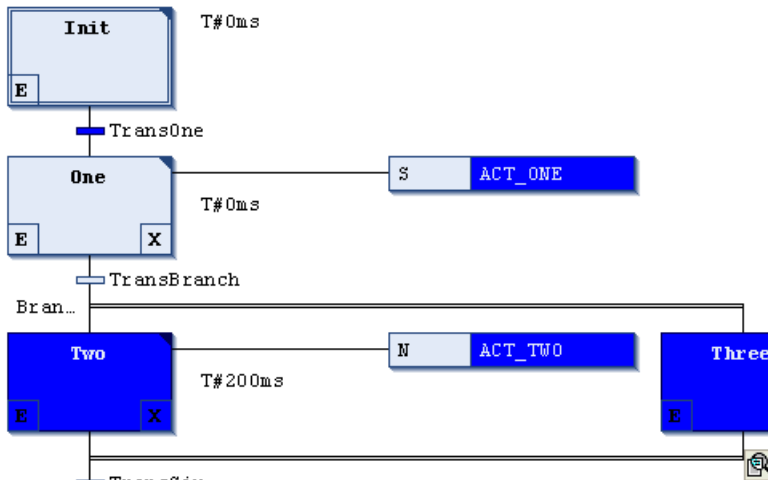
Settings regarding the online representation of the SFC elements and attributes can be made in the options of the SFC editor.

If you have explicitly declared SFC flags, in online mode they are displayed in the declaration part. In offline mode they are not displayed.



Note the processing order of the elements in an SFC diagram.

In online mode TwinCAT displays active steps in blue.



See also:

- [Implicit Variables](#) [▶ 638]
- [Processing Order in SFC](#) [▶ 636]

16.1.4.3 Processing Order in SFC

Basic behavior of the elements

- **Active step:** An active step is a step whose actions are currently being executed. In online mode TwinCAT displays active steps in blue.
- **Initial step:** In the first cycle after calling an SFC POU, the initial step is automatically activated and the step action is executed.
- TwinCAT executes the IEC actions at least twice. The first time after the step has been activated, the second time - but only in the next cycle - after the step has been deactivated.
- **Alternative branches:** If the step preceding the branch is active, TwinCAT evaluates the first transition of each alternative branch from left to right. In the first branch, in which TwinCAT finds a transition that returns TRUE, the next step is activated.
- **Parallel branches:** If the step preceding the branch (before the horizontal double line) is active and the transition before the branch returns TRUE, TwinCAT activates the first steps in all branches. The branches are then processed simultaneously. The step after the end of the branch (after the horizontal double line) is activated when all last steps in the branches are active and the transition after the double line returns TRUE.

Processing sequence

1. Resetting of the IEC actions

TwinCAT resets the internal action control flags of the action qualifiers (N, R, S, L, D, P, SD, DS, SL). These are the flags that control an IEC action. However, flags that are called within actions will not be reset!

2. Executing exit actions

TwinCAT then checks all steps to see whether they fulfill the condition for executing the exit actions. The checking sequence corresponds to the order in the SFC diagram: from top to bottom and from left to right.

TwinCAT executes an exit action when the step is deactivated, i.e. when its input and step actions (if available) have been executed in the previous cycle and the condition for the subsequent step returns TRUE.

3. Executing entry actions

TwinCAT then checks all steps to see whether they fulfill the condition for executing the entry actions. The checking sequence corresponds to the order in the SFC diagram: from top to bottom and from left to right. If this is the case, TwinCAT executes the entry actions.

TwinCAT executes an entry action as soon as processing has reached the transition preceding the step and this transition returns TRUE, i.e. the step is activated.

4. Time check, executing step actions

TwinCAT executes the following for all steps in the sequence in which they are arranged in the SFC diagram:

- TwinCAT copies the elapsed time of the active step into the corresponding implicit step variable <step name>.t.
- In the event of a timeout, TwinCAT sets the corresponding error flags.
- For non-IEC steps, TwinCAT now executes the step action.

5. Executing IEC actions

TwinCAT executes the IEC actions in alphabetical order. This is done in two cycles through the list of actions. In the first run TwinCAT executes the IEC actions of all steps that were deactivated in the previous cycle. In the second run, the IEC actions of all steps that are active are executed.

6. Transition check, activation of the following steps

The transitions are evaluated: If a step is active in the current cycle and the following transition returns TRUE (and a minimum step time that may have been defined has elapsed), the following step is activated.



When implementing actions, note the following:

It may happen that an action is executed several times within the same cycle, because it is associated with several processes. (Example: An SFC contains the two IEC actions A and B, which are both programmed in SFC and which both call an IEC action C. In this case, A and B can both be active in IEC actions in the same cycle and IEC action C can be active in both actions. In this case C would be called twice.)

If the same IEC action is used simultaneously at different levels of an SFC diagram, this can lead to unpredictable effects during execution. Therefore, a corresponding error message is issued. These may appear when working with projects that were created with an older version of the programming system.



Note the possibility of using implicit variables to monitor or control the processing status of steps and actions.

See also:

- [Implicit variables](#) [▶ 638]
- [Qualifier for actions in SFC](#) [▶ 637]

16.1.4.4 Qualifiers for Actions in SFC

You assign IEC steps to qualifiers. Qualifiers describe how an action associated with the step is executed.

The qualifiers are processed by the function block `SFCActionControl` of the system library. The library is automatically integrated into the project by the SFC plug-ins.

Available qualifiers:

N	Non-stored	The action is active as long as the step is active.
R	overriding Reset	The action is deactivated.
S	Set (Stored)	TwinCAT executes the action as soon as the step becomes active. The action continues to be executed, even if the step has already been deactivated, until it is reset.
L	time Limited	TwinCAT executes the action as soon as the step becomes active. The action is executed until the step becomes inactive or the specified timespan has elapsed.
D	time Delayed	TwinCAT does not start execution of the action until the given delay time has elapsed after the step has become active and the step is still active. The action is executed until the step is deactivated.
P	Pulse	TwinCAT executes the action precisely twice: once when the step becomes active and again in the subsequent cycle.
SD	Stored and time Delayed	TwinCAT does not start execution of the action until the specified delay time has elapsed after the step has become active. The action is executed until it receives a reset.
DS	Delayed and Stored	TwinCAT does not start execution of the action until the given delay time has elapsed after the step has become active and the step is still active. The action is executed until it receives a reset.
SL	Stored and time limited	TwinCAT executes the action as soon as the step is activated. It is executed until the specified time has elapsed or a reset is received.

You must specify the time specifications for the qualifiers L, D, SD, DS and SL in the format of a TIME constant.



When an IEC action is deactivated, it is executed one more time. This means that TwinCAT executes such an action at least twice. This also concerns actions with the qualifier P.

See also:

- [Programming in Sequential Function Chart \(SFC\) \[► 124\]](#)

16.1.4.5 Implicit Variables

Each SFC object provides implicit variables, which you can use to monitor the status of steps and IEC actions at runtime. TwinCAT automatically creates these implicit variables for each step and IEC action.

The implicit variables are structure instances of type SFCStepType (for steps) or SFCActionType (for actions). The variables have the name of the element, e.g. "step 1" for a step with step name "step1". The structure components describe the status of a step or action or the time that has already elapsed in an active step.

Step and action status

Syntax for the implicit variable declaration:

```
<stepname>:SFCStepType;
```

```
_<actionname>:SFCActionType;
```

The following implicit variables are available for step or IEC action status:

Step	
<stepname>.x	Shows the activation status in the current cycle. If <stepname>.x = TRUE, TwinCAT executes the step in the current cycle.
<stepname>._x	Shows the activation status for the next cycle. If <stepname>._x = TRUE and <stepname>.x = FALSE, TwinCAT executes the step in the next cycle, i. e. <stepname>._x is copied to <stepname>.x at the beginning of a cycle.
<stepname>.t	The flag t returns the current timespan that has elapsed since the step became active. This applies only to steps, regardless of whether a minimum time is defined in the step properties or not. See also SFC flag SFCErrror.
<stepname>._t	Used only for internal purposes
IEC action	
_ <actionname>.x	TRUE if the action is executed.
_ <actionname>._x	TRUE if the action is active.



You can use the variables described above to force a particular status value for a step, that is, to set a step to active. Note, however, that this causes an uncontrolled status of the SFC.

Access to implicit variables

Syntax for the access:

Within the POU, you assign the implicit variable directly: <variable name>:=<step name>.<implicit variable> or <variable name>:=_
<action name>.<implicit variable>

Example:

```
status := step1._x;
```

From another function block with a POU name: <variable name>:=<POU name>.<step name>.<implicit variable> or <variable name>:=<POU name>._<action name>.<implicit variable>

Example:

```
status := SFC_prog.step1._x;
```

See also:

- [SFC Element Properties \[▶ 650\]](#)
- [SFC Flags \[▶ 639\]](#)

16.1.4.6 SFC Flags

SFC flags are implicitly generated variables with predefined names. You can use them to influence the processing of an SFC diagram. You can use these flags to indicate timeouts or reset step sequences, for example. You can also activate inching mode, for example, to switch transitions selectively. To have access these variables, you must declare and activate them.

SFC-Flags

Name	Data type	Description
SFCInit	Bool	TRUE: TwinCAT resets the sequence to the initial step. The other SFC flags are also reset (initialization). As long as the variable is TRUE, the initial step remains set (active), but is not executed. The function block only continues to be processed normally when you set SFCInit back to FALSE.
SFCReset	Bool	Behaves similarly to SFCInit. In contrast, however, TwinCAT continues to process the initial step after initialization. For example, you could set the SFCReset-yyyyy flag to FALSE directly in the init step.
SFCError	Bool	Becomes TRUE if a timeout has occurred in an SFC diagram. If a further timeout occurs in the program after the first timeout, this is no longer registered if you have previously not reset the variable SFCError. The declaration of SFCError is a prerequisite for the functioning of the other flag variables to control the timing (SFCErrorStep, SFCErrorPOU, SFCQuitError).
SFCEnableLimit	Bool	Used for specific activation (TRUE) and deactivation (FALSE) of the timeout control in steps by SFCError. When declaring and activating this variable (SFC settings), you must also set it to TRUE for SFCError to work, otherwise timeouts will be ignored. This can be useful, for example, during commissioning or manual operation. If you do not declare the variable, SFCError works automatically. A prerequisite is the declaration of SFCError.
SFCErrorStep	String	Saves the name of the step that caused a timeout registered by SFCError. The name is stored until the registered timeout is reset by SFCQuitError. Prerequisite is the declaration of SFCError.
SFCErrorPOU	String	In case of a timeout, saves the name of the function block in which a timeout registered by SFCError occurred. The name is stored until the registered timeout is reset by SFCQuitError. Prerequisite is the declaration of SFCError.
SFCQuitError	Bool	As long as this Boolean variable is TRUE, TwinCAT suspends the execution of the SFC diagram. A possible timeout, stored in the variable SFCError, will be reset. If you reset the variable to FALSE, all previous times in the active steps are reset. A prerequisite is the declaration of SFCError.
SFCPause	Bool	As long as this variable is TRUE, TwinCAT suspends the execution of the SFC diagram.
SFCTrans	Bool	Becomes TRUE when a transition occurs.
SFCCurrentStep	String	Shows the name of the currently active step, regardless of the time monitoring. In a parallel branch, the name of the step of the furthest right branch is always stored.
SFCTip, SFCTipMode	Bool	Allow "tip mode" for the SFC function block. If you activate tip mode with SFCTipMode=TRUE, you can only switch to the next step by setting SFCTip to TRUE. As long as SFCTipMode is set to FALSE, you can also switch via the transitions.
SFCErrorAnalyzati on		Contains, in the form of a string, all variables that contribute to the total value TRUE of SFCError (timeout in one step). SFCError must be enabled for this purpose. SFCErrorAnalyzation implicitly uses the function block AnalyzeExpression from the Tc2_System library.
SFCError AnalyzationTable		Contains, in the form of a table, the variables that contribute to the total value TRUE of SFCError (timeout in one step). SFCError must be enabled for this purpose. SFCErrorAnalyzationTable implicitly uses the function block AnalyzeExpressionTable from the Tc2_System library.

Implicit creation of SFC flags

TwinCAT declares SFC flags automatically if you have activated the corresponding option. You can set this option in the **Properties** view for a single SFC POU or in the project properties in the **SFC** category for all SFC POUs in the project.



The SFC settings of the SFC flags of individual POUs are only effective if you have not activated the **Use default SFC settings** option. If you have activated this option, the settings defined in the project properties apply.



SFC flags that you have declared in the SFC settings dialog are only visible in the online view of the SFC function block!

Explicit creation of SFC flags

Manual declaration is only required to enable write access from another function block. In this case, note the following: if you have declared the flag in a global variable list, you must deactivate its "Declare" setting in the SFC settings dialog. Otherwise a local SFC flag is implicitly declared, which TwinCAT then uses instead of the global variables!

Application example for SFCErrror

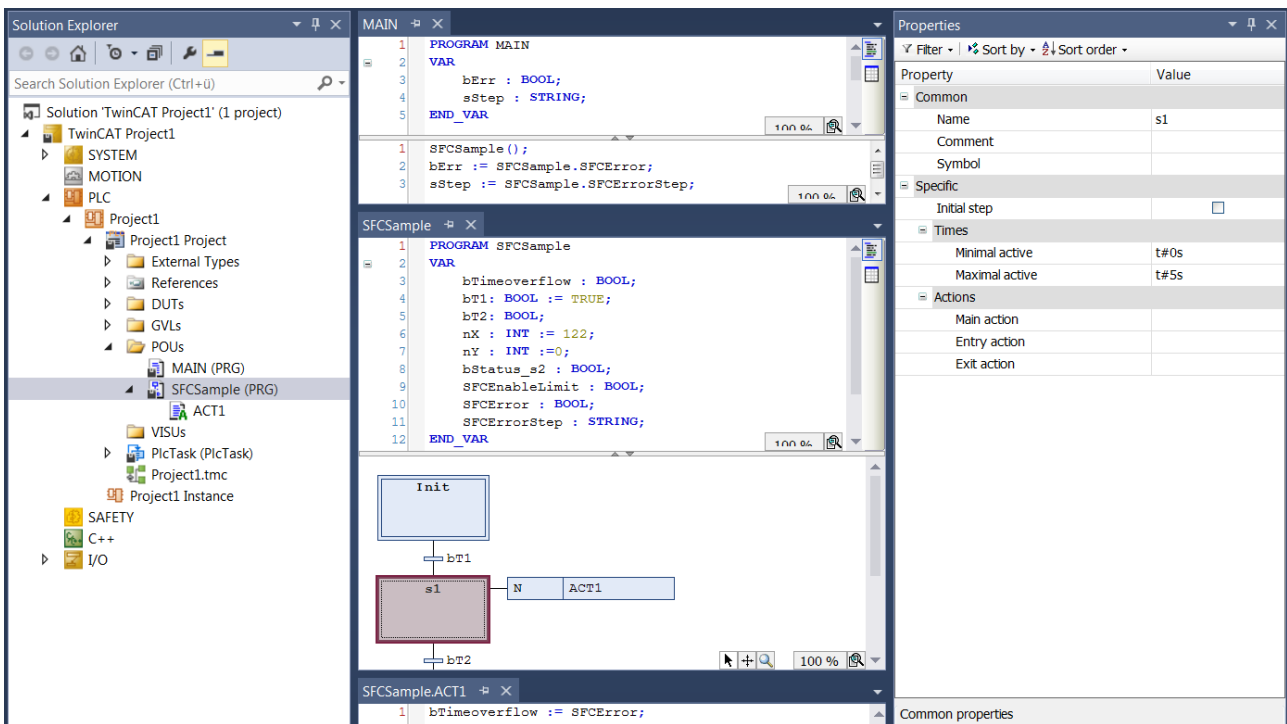
You have created an SFC function block called "SFCSample" that contains a step "s1". You have defined time limits in the step properties. See figure "Online view of SFC function block SFCSample".

If, for some reason, step s1 remains active longer than permitted in its time properties (timeout), TwinCAT sets the SFC flag `SFCErrror`, which the PLC program can access.

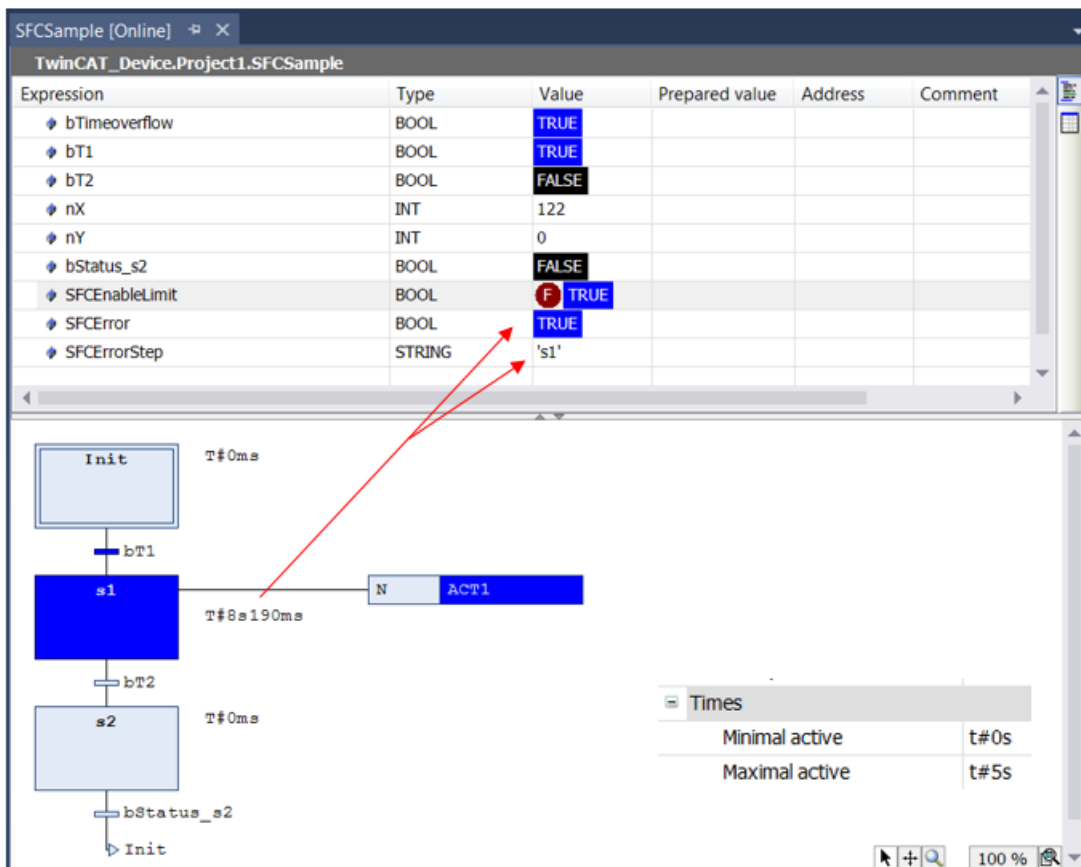
To allow this access, you must activate and declare the SFC flag in the SFC settings. If you only declare it, the SFC flag is displayed in the online view of `SFCSample` in the declaration part, but it has no function.

Solution options			
Flags			
Use	Variable	Declare	Description
<input type="checkbox"/>	SFCInit	<input checked="" type="checkbox"/>	All steps and actions are reset. The init step is activated. No actions will be executed.
<input type="checkbox"/>	SFCReset	<input checked="" type="checkbox"/>	All steps and actions are reset. The init step is activated and it's actions will be executed.
<input checked="" type="checkbox"/>	SFCErrror	<input checked="" type="checkbox"/>	Gets 'TRUE', if a time check failed.
<input checked="" type="checkbox"/>	SFCEnableLimit	<input checked="" type="checkbox"/>	Enable time check on steps
<input checked="" type="checkbox"/>	SFCErrrorStep	<input checked="" type="checkbox"/>	Contains the name of the step that caused SFCErrror to be 'TRUE'. SFCErrror is required.

Now you can address the SFC flag within the function block, for example in an action, or from outside the function block.



Online view of SFC function block SFCSample:



SFCError becomes TRUE if a timeout occurs within SFCSample.

Note the possibility of using the flags SFCErrorAnalyzation and SFCErrorAnalyzationTable to determine the components of the expression that contributes to the value TRUE of the SFCError.

Accessing the flags

Syntax for the access:

Within the POU, you assign the flag directly: <variable name>:=<SFC flag>

Example:

```
checkerror:=SFCerror;
```

From another function block with a POU name: <variable name>:=<POU name>.<SFC flag>

Example:

```
checkerror:=SFC_prog.SFCerror;
```

If you require write access from another block, you must also declare the SFC flag explicitly as a VAR_INPUT variable in the SFC function block or globally in a GVL.

Example:

Local declaration:



```
PROGRAM SFC_prog
VAR_INPUT
    SFCinit:BOOL;
END_VAR
```

or global declaration in a global variable list:

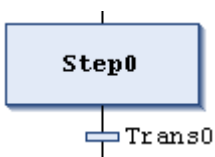
```
VAR_GLOBAL
    SFCinit:BOOL;
END_VAR

PROGRAM PLC_PRG
VAR
    setinit: BOOL;
END_VAR
SFC_prog.SFCinit:=setinit; //Schreibzugriff auf SFCinit in SFC_prog
```

16.1.4.7 Elements**16.1.4.7.1 SFC elements step and transition**

Step symbol:  , Transition symbol: 

TwinCAT always inserts steps and transitions as a combination. Inserting a step without a transition or a transition without a step results in a compile error. You can change the name by double-clicking on it.



Step names must be unique within the scope of the "father" function block. Keep this in mind when using actions that are also programmed in SFC.

Note that you can make a step an initial step with the **Init step** command or by setting the corresponding property in the SFC element Properties.

Each step is defined by the step properties, which you can view and edit depending on the options set in the **Properties** view.

You must add actions that you want to execute when the step is active. There are "IEC actions" and "Step actions". For details, refer to the **Action** section for the SFC element.




It is the user's responsibility to assign the desired expression to a transition variable if the transition contains multiple instructions.

Transitions consisting of a transition or property object are indicated by a small triangle in the upper right corner of the transition rectangle.

See also:

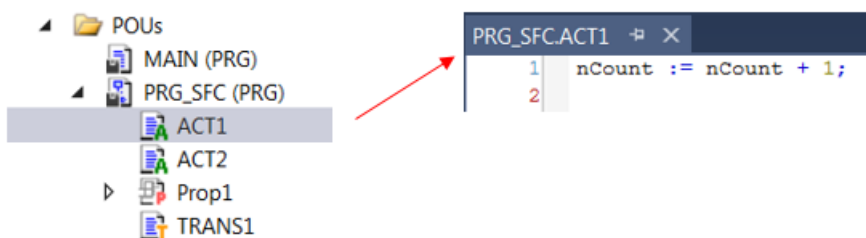
- [Programming in Sequential Function Chart \(SFC\) \[▶ 124\]](#)
- [Object Transition \[▶ 88\]](#)
- [SFC Element Properties \[▶ 650\]](#)
- [SFC element Action \[▶ 645\]](#)
- [Method call - virtual function call \[▶ 199\]](#)
- TC3 User Interface documentation: [Command Insert step transition \[▶ 1001\]](#)
- TC3 User Interface documentation: [Command Init step \[▶ 1001\]](#)

16.1.4.7.2 SFC element Action

Symbol: 

An action contains one or more instructions in one of the valid programming languages. You can assign an action to a step.

Actions that you use in SFC steps must be created as function blocks in the project.



Exception: in the case of IEC actions that you add to a step as an action association, you can also specify a boolean variable instead of an action object. The value of this variable is toggled between FALSE and TRUE each time the action is executed.

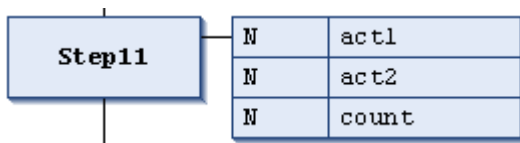
i Step names must be defined unambiguously within the scope of the "father" function block. An action written in SFC must not contain a step that has the same name as the step to which the action is assigned.

There are "IEC actions" and "Step actions".

1. IEC Actions

IEC actions are actions according to the IEC 61131-3 standard. They are executed according to their qualifier. IEC actions are executed at least twice: the first time when the step becomes active and a second time when it is deactivated. If you assign more than one action to a step, the action list is processed from top to bottom.

Each action box contains the qualifier in the first column and the action name in the second column. Both can be edited directly.



Unlike for step actions, you can use different qualifiers for IEC actions. Another difference compared with step actions is that each IEC action has a control flag. This means that TwinCAT executes the action only once, even if it is called by another step at the same time. This cannot be guaranteed for step actions.

You can assign IEC actions to a step using the **Insert action association** command in the **SFC** menu.

● Associated Boolean variables

I An associated Boolean variable is set or reset on each call of the SFC function block. This means that it is reassigned either the value TRUE or FALSE each time, regardless of whether the associated step is active or not.

If the same global boolean variable is associated as an IEC action in different SFC function blocks, this can lead to undesirable overwrite effects.

See also:

- TC3 User Interface documentation: [Command Insert action association \[► 1004\]](#)
- [Qualifiers for Actions in SFC \[► 637\]](#)

2. Step actions

These are actions that you can use to extend the IEC standard.

- **Input action:**
TwinCAT executes this action once after the step has been activated and before the main action is executed.
You reference a new action or an action already created under the SFC object from a step via the element property **Entry action (Step entry)**. You can also add a new action to the step using the **Add entry action**. The entry action is indicated by an "E" in the lower left corner of the step box.
- **Main action:**
TwinCAT executes this action after the step has been activated and any entry action has already been executed. However, unlike an IEC action (see above), it is not executed a second time when the step is deactivated again. Also, you cannot use qualifiers here.
To add an existing action, use the element property **Main action (Step active)**. A main action is indicated by a filled triangle in the upper right corner of the step box.
- **Exit action:**
TwinCAT executes this action once if the step has been deactivated. Note, however, that the execution no longer takes place in the same cycle, but at the beginning of the next one!
You reference a new action or an action already created under the SFC object from a step via the element property **Exit action (Step exit)**. You can also add a new action to the step using the **Add exit action**. The exit action is indicated by "X" in the lower right corner of the step box.

Property	Value
+ Common	
- Specific	
Initial step	<input type="checkbox"/>
+ Times	
- Actions	
Step active	act_step
Step entry	act_entry
Step exit	act_exit

Action definitions

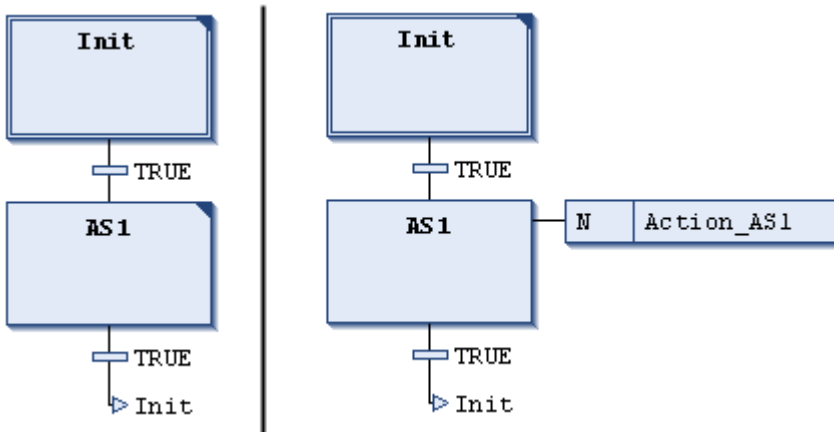
See also:

- [SFC Element Properties \[► 650\]](#)

Difference between IEC action - step action

Version

The main difference between step actions and IEC actions with qualifier "N" is that the IEC action is executed (at least) twice: the first time when the step is active and a second time when it is deactivated. See the following example:



You have added the Action_SFC1 action to step SFC1 as a step action (left), and also as an IEC action with qualifier N. Since in both cases two transitions are triggered, it takes two PLC cycles until the initialization step is reached again. Assume that Action_SFC1 incremented a counter variable nCounter that was initialized with 0. After activating the Init step again, nCounter has the value 1 in the left example. In the right example, however, it has a value of 2, since the IEC action is executed a second time due to the deactivation of SFC1.

Duplication

Another difference is that step actions can be "embedded". In this case, they can only be called from the relevant step. When you copy this step, TwinCAT automatically generates new action objects and copies the implementation code.

You can define whether a step action is "embedded" either when you insert the first action in the step, or using the step property **Duplicate or copy**. In general, this behavior can also be preset in the TwinCAT options in the **SFC editor** category.

Boolean variable

In addition, a boolean variable can be specified instead of an action object for IEC actions. This is not possible with step actions.

See also:

- [SFC Element Properties \[► 650\]](#)

16.1.4.7.3 SFC element Branch

Symbol: 

Branches are used for programming parallel or alternative sequences in SFC.

For alternative branches, TwinCAT always executes only one of the branches, depending on the previous transition condition. Parallel branches are executed simultaneously.

See also:

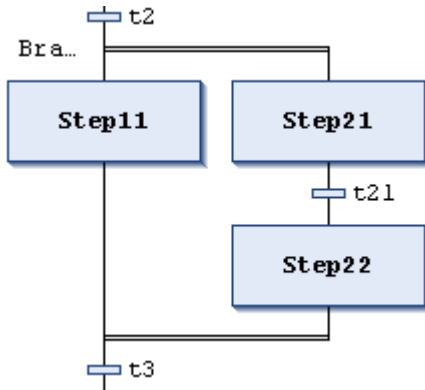
- [Programming in Sequential Function Chart \(SFC\) \[► 124\]](#)
- [Processing order in SFC \[► 636\]](#)

- TC3 User Interface documentation: [Command Insert branch right \[►_1003\]](#)

Parallel branch

In a parallel branch, the branches must begin and end with steps. Parallel branches may contain further branches.

The horizontal line before and after the branch is a double line.



Processing in online mode: If the previous transition (t2 in the example shown) is TRUE, the first steps are active in all parallel branches (Step11 and Step21). TwinCAT processes the individual branches in parallel and only then evaluates the subsequent transition (t3).

A label Branch <n> is automatically added to the horizontal line that forms the beginning of a branch. You can specify this mark as a jump destination.

Note that you can use the **Alternative** command to convert a parallel branch into an alternative branch.

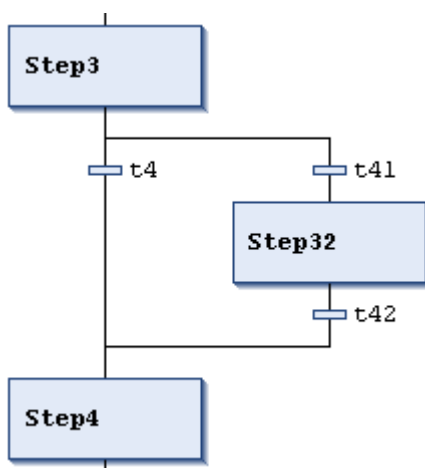
See also:

- TC3 User Interface documentation: [Command Alternative \[►_1002\]](#)

Alternative branch

The horizontal line before and after the branch is a single line.

In an alternative branch, the branches must begin and end with transitions. The branches may contain further branches.



If the step preceding the branch is active, TwinCAT evaluates the first transition of each alternative branch from left to right. The corresponding branch is "opened" in the first transition that returns TRUE, i. e. the step following the transition is activated.

Note that you can use the **Parallel** command to convert an alternative branch into a parallel branch.

See also:

- TC3 User Interface documentation: [Command Parallel](#) [▶ 1002]

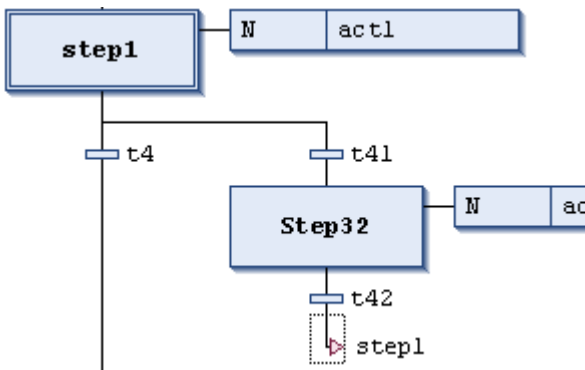
16.1.4.7.4 SFC element Jump

Symbol: 

A jump defines which step is to be executed next as soon as the transition preceding the jump becomes TRUE. Jumps may be necessary because the execution lines may not cross and cannot lead upwards.

In addition to the mandatory jump at the end of the diagram, you can only enter jumps at the end of a branch.


The destination of a jump is defined by the added text string, which you can edit directly. The jump destination can be a step name or the mark of a parallel branch.



See also:

- [Programming in Sequential Function Chart \(SFC\)](#) [▶ 124]
- TC3 User Interface documentation: [Command Insert jump](#) [▶ 1005]

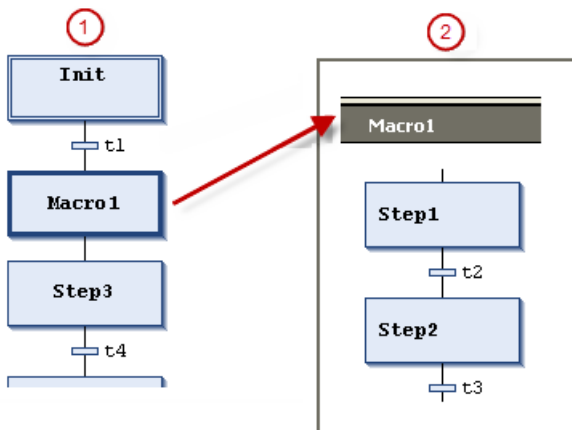
16.1.4.7.5 SFC element Macro

Symbol: 

A macro contains a part of the SFC diagram, which is not shown in detail in the main view of the editor.

The processing flow is not affected by the use of macros. Macros are only used to hide certain parts of the diagram in the interest of clarity, for example.

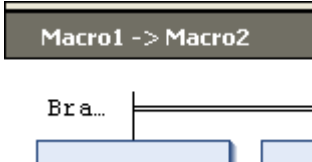
Open the macro editor by double-clicking on the macro box or with the command **Show macro** in the **SFC** menu. You can program here in the same way as in the main view of the SFC editor. To close the macro editor, use the **Exit macro** command in the **SFC** menu.



- ① Main view in the SFC editor
- ② View in the macro editor for Macro1

Macros can contain further macros. The title bar of the macro editor always shows the path of the currently open macro within the diagram.

Example:



See also:

- [Programming in Sequential Function Chart \(SFC\) \[► 124\]](#)
- TC3 User Interface documentation: [Command Show macro \[► 1007\]](#)
- TC3 User Interface documentation: [Command Exit macro \[► 1007\]](#)

16.1.4.7.6 SFC Element Properties

You can edit the properties of an SFC element in the **Properties** view. Use the command **View > Properties Window** to open this view. Which properties are displayed depends on the currently selected element.



Which properties are displayed next to the element in the SFC diagram depends on the TwinCAT options settings in category **TwinCAT > PLC Environment > SFC editor, View** tab.

General

Property	Value description
Name	Element name, default <element><consecutive number>, e.g. step name "Step0", "Step1", branch name "Branch0", etc.
Comment	Element comment (text), e.g. counter reset. You can insert line breaks with [Ctrl] + [Enter] .
Symbol	For each SFC element, TwinCAT creates an implicit variable with the same name as the element.

Specific

Property	Value description
Init step	<p><input checked="" type="checkbox"/> : This option is only enabled for the step that is currently defined as the initial step. By default, this is the first step in an SFC diagram.</p> <p>If you have activated this property for another step, it must be deactivated for the step that previously had this property, in order to avoid compile errors.</p>
Duplicate or copy	<p>This option is available for steps that contain a step action (entry action, main action or exit action), and for transitions that are linked to a transition object.</p> <p><input checked="" type="checkbox"/> : When you copy the step/transition, a new object is created for each of the called actions/transitions. It contains a copy of the implementation code of the copied object.</p> <p><input type="checkbox"/> : When you copy the step or transition, the link to the relevant object is retained for the associated actions/transitions. No new objects are created. The source and copies of the step or transition call the same action/transition.</p> <p>Embedded objects are displayed in the PLC project tree with an underscore in the name.</p> <p>You can use the commands Change duplication > Set and Remove to "embed" all step actions or transitions that are called in an SFC function block or to cancel the embedding.</p>
Times <ul style="list-style-type: none"> • Minimum active • Maximum active 	<p>Minimum time for which the step is active, even if the subsequent transition is TRUE.</p> <p>Maximum time for which the step may be active. If the time is exceeded, TwinCAT sets the implicit variable SFCErrror to TRUE.</p> <p>Time specifications according to the IEC syntax (e.g. t#8 s) or TIME variable; default: t#0s.</p>
Actions <ul style="list-style-type: none"> • Entry Action • Step action • Exit action 	<ul style="list-style-type: none"> • Entry action: TwinCAT executes this action after the step has been activated. • Step action: TwinCAT executes this action if the step is active and any entry action has already been executed. • Exit action: If the step is deactivated, TwinCAT performs this action in the following cycle. <p>Note the processing sequence.</p>



You can use the corresponding implicit SFC variables and flags to obtain information about the status of a step or an action, or about timeouts.

See also:

- [Implicit variables \[▶ 638\]](#)
- [TC3 User Interface documentation: Dialog Options - SFC editor \[▶ 977\]](#)

16.1.5 Function Block Diagram / Ladder / Instruction List (FBD/LD/IL)

16.1.5.1 FBD/LD/IL Editor

The FBD/LD/IL editor is a combined editor for the programming languages FBD, LD and IL.



If required, IL can be enabled via the TwinCAT options. (**Tools > Options > TwinCAT > PLC Environment > FBD, LD and IL > IL**)

There is a common set of commands and elements, and TwinCAT automatically converts the three programming languages internally.

The code in the implementation part is structured in all three languages using networks.

The **FBD/LD/IL** menu contains commands for working in the editor.

In offline and online mode, you can switch between the three editor views at any time using the command.

The behavior of the FBD/LD/IL editor is determined by the settings in the menu **Tools > Options**, category **TwinCAT > PLC Environment > FBD, LD and IL**.

i There are some special elements that TwinCAT cannot convert and thus only displays in the appropriate language. Similarly, there are constructs which cannot be unambiguously converted between IL and FBD and are therefore "normalized", i.e. canceled, when converted back into FBD. This concerns: negation of expressions and explicit/implicit assignment of function block inputs and outputs.

See also:

- [General functionalities in all graphical editors \[▶ 623\]](#)
- [TC3 User Interface documentation: FBD/LD/IL \[▶ 1025\]](#)
- [TC3 User Interface documentation: Dialog options - FBD, LD and IL \[▶ 970\]](#)

FBD and LD editor

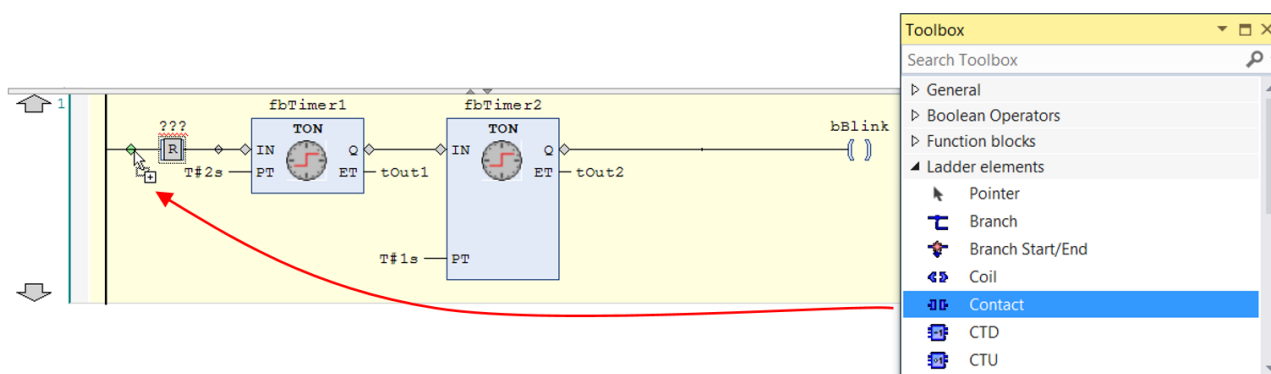
Inserting and arranging items:

You can move elements from the **Toolbox** view to the implementation part of the editor by dragging them with the mouse. Alternatively, you can use the commands of the context menu or the menu **FBD/LD/IL**.

Settings relating to display and interface are defined in the TwinCAT options, category **TwinCAT > PLC Environment > FBD, LD and IL**.

When you drag an element with the mouse over a network in the editor, all possible insertion positions are displayed with gray diamond, triangle or arrow-shaped position marks. When the mouse pointer is positioned over one of these markers, the marker turns green, and TwinCAT inserts the element at the position when the mouse button is released.

Example:



When you drag a function block or operator from the **Toolbox** or a network to the left edge of the network onto one of the two arrows, the following happens: TwinCAT automatically creates a new network and inserts the element there.

To replace an element, use the mouse to drag a suitable other element to its position. In the editor, TwinCAT identifies elements that can be replaced by the new element with text fields, e. g. "Replace" or "Append input".

To cut, copy, paste and delete elements, you can use the usual commands of the **Edit** menu. Copying also works with drag and drop while the **[Ctrl]** key is pressed.



The operators with EN/ENO functionality can only be inserted in the FBD and LD editor.

Selecting elements:

You can select a function block or connecting line in the editor by clicking on it, so that the focus is on it. Multiple selection is possible while the button **[Ctrl]** is pressed. A selected element is shaded in red.

Tooltip:

When the cursor points to specific elements, such as a variable or an input, a tooltip appears with information about that element.

For elements with a squiggly red line, the tooltip shows the corresponding precompile error message.

Navigating in the editor:

Using the buttons and commands described below, you can move the focus to a different cursor position within the editor. Switching between positions also works across networks.	
[←] [→]	Change to the adjacent cursor position, along the signal flow, i.e. from left to right and vice versa.
[↑][↓]	Change to the next cursor position above or below the current position, if this adjacent position belongs to the same logical group. A logical group is formed by all the connections of a function block, for example. If such a logical group does not exist: Change to the first cursor position in the next upper or lower adjacent element. In the case of elements connected in parallel, navigation takes place along the first branch.
[Ctrl] + [Pos 1]	Change to the first network; this is selected.
[Ctrl] + [End]	Change to the last network; this is selected.
[Image up]	Scroll up 1 page. The top network on this page is selected.
[Image down]	Scroll down 1 page. The bottom network on this page is selected.
Command "Go To..."	Change to a specific network.

Opening a function block:

If a function block is inserted in the editor, you can double-click its implementation by double-clicking or by using the command **Go To** of the context menu.

See also:

- [Function block diagram \(FBD\) \[▶ 109\]](#)
- [Programming Function Block Diagrams \(FBD\) \[▶ 110\]](#)
- [Ladder diagram \(LD\) \[▶ 109\]](#)
- [Programming Ladder Diagrams \(LD\) \[▶ 111\]](#)
- [Elements \[▶ 660\]](#)
- [FBD/LD/IL editor in online mode \[▶ 655\]](#)
- [TC3 User Interface documentation: Command Go To \[▶ 1037\]](#)

IL-Editor

Inserting and arranging items:

You can insert elements by using the commands of the menu **FBD/LD/IL** or the context menu. A new network can also be added from the **Toolbox** into the implementation part of the editor with drag & drop.

To cut, copy, paste and delete elements, you can use the usual commands of the Edit menu. Copying also works with drag and drop while the **[Ctrl]** key is pressed.



Note that operators with EN/ENO functionality can only be inserted in the FBD and LD editor.

Each program line is entered in a table row.

Structure of a network in the IL editor:

Line 1: Network title		
Requirement: The option is activated in the TwinCAT options		
Line 2: Network comment		
Requirement: The option is activated in the TwinCAT options		
from line 3:		
Column	Contents	Description
1	Operator	Contains the IL operator (LD, ST, CAL, AND, OR etc.) or a function name. If you call a function block, you must also specify the corresponding parameters here; in this case, you must enter := or => in the preceding field.
2	Operand	Contains precisely one operand or the name of a label. If there are several operands, you must enter them in several lines and insert a comma directly behind the individual operands. (See example below)
3	Address	Contains the address of the operand as defined for its declaration. not editable You can enable/disable the display using the Show symbol address option. To do this, select the command Tools > Options and in the category TwinCAT > PLC Environment > FBD, LD and IL select the General tab.
4	Symbol comment	Contains any comment that may have been entered for the operand in the declaration. not editable You can enable/disable the display using the option Show symbol comment by selecting the command Tools > Options and selecting the General tab in the category TwinCAT > PLC Environment > FBD, LD and IL .
5	Operand comment	Comment relating to the current program line. You can enable/disable the display using the option Show operand comment by selecting the command Tools > Options and selecting the General tab in the category TwinCAT > PLC Environment > FBD, LD and IL .

Example:

CAL	fbTonInst1 (
	IN:= bVar,		
	PT:= tTime1,		
	ET=> tOut1)		
LD	fbTonInst1.Q		<i>gets TRUE, delay time (PT) aft...</i>
ST	fbTonInst2.IN		<i>starts timer with rising edge,...</i>
CAL	fbTonInst2 (
	PT:= tTime2,		
	Q=> bReady,	§Q*	<i>for fbTonInst2</i>
	ET=> tOut2)		

Navigating in the editor:

Key(s)/command	Cursor movement
[↑][↓]	Jump to the field above/below.
[Tab]	Jump right to the next field within the line.
[Shift] + [Tab]	Jump left to the preceding field within the line
[Space]	Opens the edit frame for the selected field. Alternatively, you can click on the field with the mouse. If applicable, the button for the Input Assistant dialog is available.
[Ctrl] + [Enter]	Inserts a new line below the current line.
[Del]	Deletes the current line.
[Ctrl] + [Pos 1]	Sets the focus to the beginning of the document and marks the first network.
[Ctrl] + [End]	Moves the focus to the end of the document and marks the last network.
[Image down]	Scrolls up a page and marks the topmost rectangle.
[Image up]	Scrolls down a page and marks the topmost rectangle.

See also:

- [Instruction list \(IL\) \[▶ 110\]](#)
- [Programming Instruction Lists \(IL\) \[▶ 112\]](#)
- [Modifiers and Operators in IL \[▶ 656\]](#)
- [FBD/LD/IL editor in online mode \[▶ 655\]](#)

16.1.5.2 FBD/LD/IL editor in online mode

In online mode, the editor displays the current value of each variable after the variable name. Writing/forcing and setting breakpoints is possible.

If the variable is currently forced, this is indicated by **F** directly in front of the forced value. If a value is prepared for writing or forcing, this value is displayed in parentheses <value> immediately after the current value.

Example:

Forced variable:

bVar1 **F** TRUE

Prepared value

nVar1 0<10>

In the online view of a Ladder Diagram (LD), the connection cables are color-coded: Connections with the value TRUE are displayed as a bold blue lines, connections with the value FALSE as a bold black lines. Connections of unknown or analog value are displayed normally (thin black line).



Note that the value of the connections is calculated from the monitored variables. This is not a real process control.

Breakpoints

Possible locations for breakpoints are the positions where variable values can change (instructions), where the program branches, or where an another function block is called.

Possible breakpoint positions:

- On the entire network: Causes the breakpoint to be set at the first available position in the network.

- On a function block box, if the function block contains an assignment. Not possible for operator function blocks, for example ADD, DIV.
- At assignments.
- At the end of the function block at the position of the return to the calling function block. In online mode, an empty network automatically appears at this point, which is marked with "RET" instead of a network number.

i At present, you cannot set a breakpoint directly on the first function block in the network. However, if you set a breakpoint on the entire network, this breakpoint marker is automatically transferred to the first function block in online mode.

i Breakpoints in methods: TwinCAT automatically sets a breakpoint in all methods that can be called. This means that if a method managed by an interface is called, breakpoints are set in all methods that occur in function blocks that implement this interface, as well as in all derived function blocks that use the method. If a method is called via a pointer to a function block, TwinCAT sets the breakpoints in the method of the function block and in all derived function blocks that use the method.

See also:

- [Forcing and Writing Variables Values \[► 214\]](#)
- [Use of breakpoints \[► 211\]](#)

16.1.5.3 Modifiers and Operators in IL

Modifiers:

Modifier	Combined with operator	Description
C	JMP, CAL, RET	The instruction is only executed if the result of the preceding expression is TRUE.
N	JMPC, CALC, RETC	The instruction is only executed if the result of the preceding expression is FALSE.
N	otherwise	Negation of the operand (not the accumulator).

Operators with the possible modifiers:

Operator	N	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 in the operands.	ST iErg
S		Sets the operand (type BOOL) to TRUE, if the content of the accumulator is TRUE.	S bVar1
R		Sets the operand (type BOOL) to FALSE, if the content of the accumulator is TRUE.	R bVar1
AND	N,(Bitwise AND of the accumulator value and the (negated) operand	AND bVar2
OR	N,(Bitwise OR of the accumulator value and the (negated) operand	OR xVar
XOR	N,(Bitwise, exclusive OR of the accumulator value and the (negated) operand	XOR N,(bVar1,bVar2)
NOT		Bitwise negation of the accumulator value	
ADD	(Addition of the accumulator value and the operand. The result is written into the accumulator.	ADD iVar1
SUB	(Subtraction of the operand from the accumulator value. The result is written into the accumulator.	SUB iVar2
MUL	(Multiplication of the accumulator value and the operand. The result is written into the accumulator.	MUL iVar2
DIV	(Division of the accumulator value by the operand. The result is written into the accumulator.	DIV 44
GT	(Checks whether the accumulator value is greater than the value of the operand. The result (BOOL) is written into the accumulator. >	GT 23
GE	(Checks whether the accumulator value is greater than or equal to the value of the operand. The result (BOOL) is written into the accumulator.	GE iVar2
EQ	(Checks whether the accumulator value is equal to the value of the operand. The result (BOOL) is written into the accumulator.	EQ iVar2
NE	(Checks whether the accumulator value is equal to the value of the operand. The result (BOOL) is written into the accumulator.	NE iVar1
LE	(Checks whether the accumulator value is greater than or equal to the value of the operand. The result (BOOL) is written into the accumulator.	LE 5
LT	(Checks whether the accumulator value is less than the value of the operand. The result (BOOL) is written into the accumulator.	LT cVar1
JMP	CN	Unconditional (conditional) jump to the specified label	JMPN next
CAL	CN	(Conditional) call of a program or function block (if the accumulator value is TRUE)	CAL prog1
RET		Exits the block and returns to the calling function block	RET
RET	C	If the accumulator value is TRUE: Exits the function block and returns to the calling function block	RETC
RET	CN	If the accumulator value is FALSE: Exits the function block and returns to the calling function block	RETCN
)		Evaluates the deferred operation	

Example:

1	AND	TRUE	load TRUE to accumulator
	ANDN	bVar1	execute AND with negated value of bVar1
	JMPC	m1	if accum. is TRUE, jump to label "m1"
	LDN	bVar2	store negated value of bVar2...
	ST	bRes	... in bRes
2	<u>m1:</u>		
	LD	bVar2	store value of bVar2...
	ST	bRes	... in bRes

Application	Description	Example
Multiple operands for 1 operator	<p>Possibilities</p> <ul style="list-style-type: none"> You enter the operands in successive lines, separated by a comma in the 2nd column. You repeat the operator in successive lines. 	<p>Version 1:</p> <pre> 1 LD 2 ADD 3, ADD 4, ADD 6, ST nVar </pre> <p>Version 2:</p> <pre> 1 LD 2 ADD 3 ADD 4 ADD 6 ST nVar </pre>
Complex operands	<p>For a complex operand, specify the opening bracket (in the first column. Enter the closing bracket in the first column in a separate line after the operand entries for the following lines.</p>	<p>A string is rotated by one character each cycle:</p> <pre> 1 LD sRotate RIGHT (sRotate LEN SUB 1) CONCAT (sRotate LEFT 1) ST sRotate </pre>
Function block call, subroutine call	<p>Column 1: Operator CAL or CALC</p> <p>Column 2: Name of the function block instance or program and opening bracket (. If no parameters follow, the closing bracket) is entered here.</p> <p>Subsequent lines:</p> <p>Column 1: Parameter name followed by: = for input parameters or => for output parameters</p> <p>Column 2: Parameter value followed by comma, if followed by other parameters. After the last parameter the closing bracket) is entered.</p> <p>Complex expressions cannot be used here, which is a restriction with regard to the IEC standard. You must assign such constructs to the function block or the program before the call.</p>	<pre> 1 CAL ProgToCall(nCounter:= 1, nDecrement:= 1000, nError=> nResult) LD ProgToCall.bError ST bErr </pre>
Function call	<p>Line 1: Column 1: LD</p> <p>Column 2: Input Variable</p> <p>Line 2: Column 1: Function name column 2: further input parameters, separated by commas.</p> <p>TwinCAT writes the return value to the accumulator.</p> <p>Line 3: Column 1: ST column 2: Variable to which the return value is written</p>	<pre> 1 LD nVar7 F_GeomAverage 25 ST nAve </pre>
Action call	<p>Like function block or program call.</p> <p>The action name is appended to the FB instance or program name.</p>	<pre> 1 CAL ProgToCall.ResetAction </pre>

Application	Description	Example
Jump	<p>Column 1: Operator JMP or JMPC.</p> <p>Column 2: Name of the label of the destination network.</p> <p>For an unconditional jump, the preceding instruction sequence must end with one of the following commands: ST, STN, S, R, CAL, RET, JMP</p> <p>For a conditional jump, the execution depends on the loaded value.</p>	<pre> 1 LD bVar1 JMPC Label1 </pre>

See also:

- [Instruction list \(IL\) \[► 110\]](#)
- [Programming Instruction Lists \(IL\) \[► 112\]](#)

16.1.5.4 Elements**16.1.5.4.1 FBD/LD/IL element network**

Symbol: 

A network is the basic unit of an FBD or LD program. In the FBD/LD/IL editor, the networks are arranged in a list below each other. Each network has a serial number on the left-hand side and can contain the following: logical and arithmetic expressions; program, function and function block calls; a jump or return instruction.

An IL program consists of at least one network. This network can contain all IL instructions for the program.

You can assign each network a title, a comment or a mark. In the TwinCAT options, category **TwinCAT > PLC Environment > FBD, LD and IL** you can define whether the network title, network comment and separator lines between the individual networks are displayed in the editor.

To enter a network title, click on the 1st line of the network. To enter of a comment, click on the 2nd line of the network.

See also:

- TC3 User Interface documentation: [Dialog options - FBD, LD and IL \[► 970\]](#)
- TC3 User Interface documentation: [Command Insert Network \[► 1025\]](#)

16.1.5.4.2 FBD/LD/IL element function block

Symbol: 

A block and its call can represent additional functions, for example IEC function blocks, IEC functions, library blocks, operators.

A function block can have any inputs and outputs.

If the function block is associated with an image file, the function block symbol is displayed within the function block. A prerequisite is that the option **Show function block icon** is enabled in the TwinCAT options, category **TwinCAT > PLC Environment > FBD, LD and IL**.

If you have changed the function block interfaces, you can update the function block parameter with the command **Update parameters** in the menu **FBD/LD/IL**, without having to reinsert the function block.

See also:

- TC3 User Interface documentation: [Command Update parameters \[► 1035\]](#)
- TC3 User Interface documentation: [Dialog options - FBD, LD and IL \[► 970\]](#)

- TC3 User Interface documentation: [Command Insert Box \[► 1026\]](#)

16.1.5.4.3 FBD/LD/IL element assignment

Symbol: 

The FBD editor displays a new assignment as a line followed by three question marks. The LD editor displays a new assignment as a coil with three question marks above.

After the insertion you can replace the ??? placeholder with the name of the variable, to which the signal coming from the left is to be assigned. The **input assistant** is available for this purpose.




In IL, an assignment is programmed using the operators LD and ST.

See also:

- [FBD/LD/IL Editor \[► 651\]](#)
- TC3 User Interface documentation: [Command Insert Assignment \[► 1026\]](#)

16.1.5.4.4 FBD/LD/IL element function block with EN/ENO

Symbol: 

The element is only available in the FBD and the LD editor.


The function block generally corresponds to the FBD/LD/IL **function block**. In addition, this function block has an EN input and an ENO output. EN and ENO have the data type BOOL.

Function of the EN input and ENO output: If input EN has the value FALSE when the function block is called, the operations defined in the function block are not executed. Otherwise, i.e. if EN is TRUE, these operations are executed. The ENO output has the same value as the EN input.

See also:

- [FBD/LD/IL element function block \[► 660\]](#)
- TC3 User Interface documentation: [Command Insert Box with EN/ENO \[► 1027\]](#)

16.1.5.4.5 FBD/LD/IL element input

Symbol: 

The maximum number of inputs depends on the function block type.

A newly added input is initially assigned ???. You can replace the ??? string with a variable or a constant.

See also:

- TC3 User Interface documentation: [Command Insert Input \[► 1028\]](#)

16.1.5.4.6 FBD/LD/IL element label

The label is an optional identifier for a network in FBD and LD, which you can use to specify a jump destination.

When you insert a label in a network, it is added as an editable **Label:** field.

See also:

- TC3 User Interface documentation: [Command Insert label \[► 1028\]](#)

16.1.5.4.7 FBD/LD/IL element jump

Symbol: 

In FBD or LD, a jump is inserted either directly before an input, directly after an output or at the end of the network, depending on the current cursor position.

You can enter a label as the jump destination directly after the jump element.

In IL, the JMP instruction is used to program a jump.

See also:

- [FBD/LD/IL element label \[► 661\]](#)
- TC3 User Interface documentation: [Command Insert jump \[► 1027\]](#)

16.1.5.4.8 FBD/LD/IL element return

The element is used to stop the function block execution immediately when the input of the RETURN element becomes TRUE.

In an FBD or LD network you can position the return instruction parallel to or after the preceding elements.

In IL, the RET instruction is available for this purpose.

See also:

- [Modifiers and Operators in IL \[► 656\]](#)
- TC3 User Interface documentation: [Command Insert Return \[► 1028\]](#)

16.1.5.4.9 FBD/LD/IL element line branch

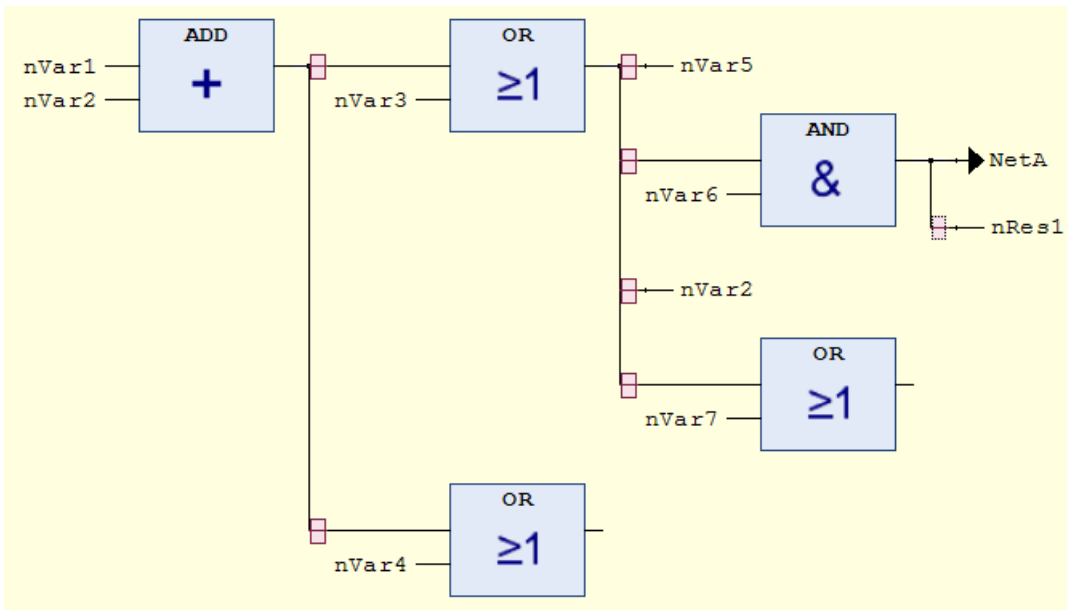
Symbol: 

The element is available in the LD and FBD editor and represents an open line branch. A line branch splits the processing line from the current cursor position into 2 subnetworks, which are executed one after the other from top to bottom. You can branch each subnetwork further, creating multiple branches within a network.

Each subnet is given a marker symbol (rectangle) at the branch point, which you can select to execute additional commands.



The Copy, Cut and Paste commands are not available for subnetworks.



To delete a subnet, first delete all elements of the network, then delete the subnet marker symbol.

See also:

- TC3 User Interface documentation: [Command Insert Branch |> 1034](#)
- TC3 User Interface documentation: [Command Insert Branch above |> 1034](#)
- TC3 User Interface documentation: [Command Insert Branch below |> 1034](#)

16.1.5.4.10 FBD/LD/IL element execute

Symbol:

The element is a function block which is used to enter ST code directly in the FBD and LD editor.

You can use the mouse to drag the **Execute** element from the **Toolbox** view into the implementation part of your POU. Clicking on the field **Enter ST Code here ...** within the function block opens an input field where you can enter multi-line ST code.

16.1.5.4.11 LD element contact

Symbol: , in the editor:

The element is only available in the LD editor.

A contact passes the signal TRUE (ON) or FALSE (OFF) from left to right, until the signal finally reaches a coil in the right part of the network. To this end, the contact is assigned a Boolean variable that contains the signal. To do this, replace the **???** placeholder above the contact with the name of a Boolean variable.

You can arrange several contacts in series or in parallel. If two parallel contacts are used, only one of them has to have the value TRUE for ON to be passed to the right. If contacts are connected in series, all contacts must be set to TRUE for ON to be passed to the right from the last contact of the series. You can therefore use LD to program electrical parallel and series circuits.


A negated contact passes on the signal TRUE if the variable value is FALSE. You can add an inserted contact by using the command **Negation** or a negated contact from the **Toolbox** view.

If you put the cursor on a contact in the selected network and press the left mouse button, the button **Convert to Coil** appears in the network. If you now move the cursor to this button with the mouse button held down and release the mouse button over the button, TwinCAT converts the contact into a coil.

See also:

- TC3 User Interface documentation: [Command Negation \[► 1032\]](#)
- TC3 User Interface documentation: [Command Insert Contact \[► 1030\]](#)
- TC3 User Interface documentation: [Command Insert Negated Contact \[► 1031\]](#)
- TC3 User Interface documentation: [Command Insert Contact \(right\) \[► 1025\]](#)
- TC3 User Interface documentation: [Command Insert Contact Parallel \(above\) \[► 1030\]](#)
- TC3 User Interface documentation: [Command Insert Contact Parallel \(below\) \[► 1030\]](#)


16.1.5.4.12 LD element coil

Symbol: , in the editor: `()`



The element is only available in the LD editor.

A coil takes the value delivered from the left and stores it in the Boolean variable assigned to it. Its input can have the value TRUE (ON) or FALSE (OFF).

Several coils in a network can only be arranged in parallel.

In a negated coil , the negated value of the incoming signal is stored in the Boolean variable assigned to the coil.

Set coil and Reset coil

Symbol:  , , in the editor: `(S)` , `(R)`

Set coil: If the value TRUE arrives at a set coil, the coil retains the value TRUE. As long as the PLC program is running, the value at this point can no longer be overwritten.

Reset coil: If the value TRUE arrives at a reset coil, the coil retains the value FALSE. As long as the PLC program is running, the value at this point can no longer be overwritten.

You can define an inserted coil as a set or reset coil by using the command **Set/Reset** in the menu **FBD/LD/IL** or insert it as a set coil or reset coil from the **Tools** view.

See also:

- TC3 User Interface documentation: [Command Set/Reset \[► 1033\]](#)
- TC3 User Interface documentation: [Command Insert Coil \[► 1029\]](#)
- TC3 User Interface documentation: [Command Insert Reset coil \[► 1029\]](#)

16.1.5.4.13 LD element line branching start/end

Symbol: 

The element is used for a closed line branch.

See also:

- [Closed line branch \[► 665\]](#)
- TC3 User Interface documentation: [Command Set Branch Start Point \[► 1034\]](#)
- TC3 User Interface documentation: [Command Set Branch End Point \[► 1035\]](#)

16.1.5.4.14 Closed line branch

A closed line branch is only available in LD. It contains a start point and an end point. It is used to implement parallel evaluation of logical elements.

Inserting a closed line branch:

- TC3 User Interface documentation: [Command Insert Contact Parallel \(below\) \[► 1030\]](#)
- TC3 User Interface documentation: [Command Insert Contact Parallel \(above\) \[► 1030\]](#)
- TC3 User Interface documentation: [Command Set Branch End Point \[► 1035\]](#)
- TC3 User Interface documentation: [Command Set Branch Start Point \[► 1034\]](#)

Closed line branch at a contact

If you have marked one or more contacts and you execute an **Insert Contact Parallel** command, a parallel branch with a simple vertical line is inserted. In this type of branch, the signal flow passes through both branches. It is an OR construct for the two branches.

Closed line branch on a function block

If you have marked a function block and you execute an "Insert contact parallel" command, a parallel branch with a double vertical line is inserted. This indicates that short-circuit evaluation (SCE) is implemented. SCE allows the execution of a function block with a Boolean output to be bypassed if a certain condition is TRUE. The condition can be represented in the LD editor through a branch that is arranged parallel with the function block branch. The short-circuit condition is defined through one or several contacts in this branch, which are connected in parallel or sequentially.

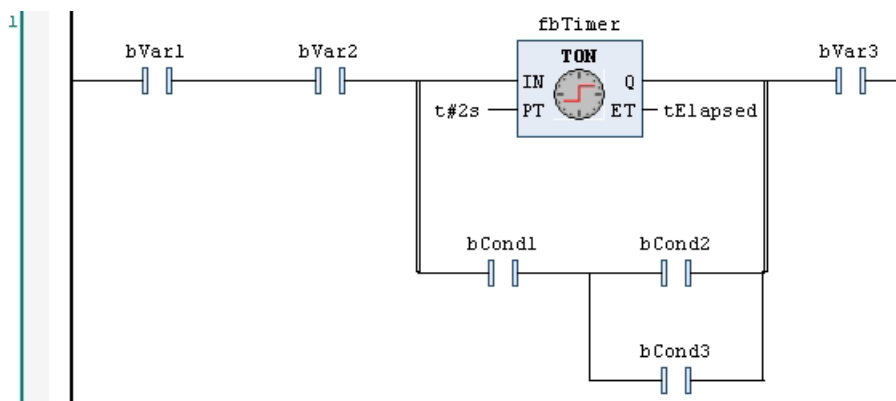
Functioning:

First, the branches which do not contain the function block are considered. If TwinCAT determines the value TRUE for one of these branches, the function block is not called in the parallel branch. In this case, the value that is applied to the input of the function block is immediately transferred to the output. If TwinCAT determines FALSE for the short-circuit evaluation condition, the function block is called and the Boolean result of its processing is passed on. If all branches contain function blocks, they are processed from top to bottom, and their outputs are linked with logical OR operations. If there are no branches with function blocks, normal OR operations are performed.

Example:

The function block instance fbTimer (TON) has a Boolean input and a Boolean output. The execution of fbTimer is omitted if TRUE is determined for the condition in the parallel line branch. The condition value results from the OR and AND operations, which link the contacts bCond1, bCond2 and bCond3.

fbTimer is executed if the conditional value from the link of the contacts bCond1, bCond2 and bCond3 is FALSE.



(1) Connections shown with a double vertical line indicate a construct that is subject to short-circuit evaluation.

(2) Connections shown with a single vertical line indicate an OR construct.

The LD example introduced above is shown as ST code below. bIN and bOUT are the Boolean values at the input (split point) and output (reunite point) of the parallel line branch.

```
bIN := bVar1 AND bVar2;

IF ((bIN AND bCond1) AND (bCond2 OR bCond3)) THEN
  bOUT := bIN;
ELSE
  fbTimer(IN := bIN, PT := {p 10}t#2s);
  tElapsed := fbTimer.ET;
  bOUT := fbTimer.Q;
END_IF

bRes := bOUT AND bVar3;
```

16.1.6 Continuous Function Chart (CFC) and Page-Oriented CFC

Viewed from the outside, a function block diagram consists of inputs and outputs between which data is processed. Internally, a function chart consists of blocks and their connections that represent data (signals) and show how allocation operators function in ST. The overall behavior is composed of the behavior of the inserted function blocks that call other programming blocks or library blocks.

Code in the Continuous Function Chart (CFC) implementation language primarily illustrates the flow of data through the system. A Continuous Function Chart is therefore also referred to as a block diagram.

In the page-oriented CFC editor, you can connect programming blocks together and create descriptive function charts distributed over pages. The page-oriented editor behaves like the CFC editor with additional functionality:



- Create pages
- Setting the page size
- Copying and pasting pages in the page navigator
- Copying the implementation of a programming block of the implementation language CFC and pasting it into a page
- Clear and space-saving arrangement of inputs, outputs and connection marks in the margins
- Connection across pages with connection marks

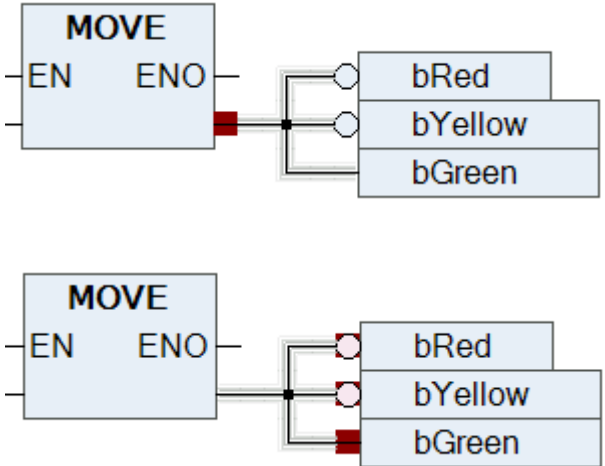
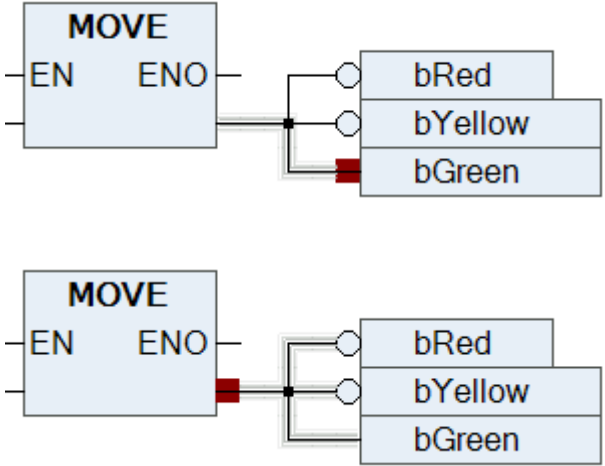
16.1.6.1 CFC Editor

Configuring the editor

You can configure the appearance, behavior and printing project-wide in the TwinCAT options in the category **TwinCAT > PLC Programming Environment > CFC Editor**. For example, in the **View** tab you can configure the color of the connecting lines depending on the data type.

Edit

Cursor symbol 	<p>Requirement: in the Toolbox view Pointer is selected.</p> <p>The icon indicates that you can edit in the editor. Select elements or connections to move them or to select commands.</p>
Cursor symbol	<p>Requirement: an element is selected in the Toolbox view.</p> <p>With a mouse click in the editor the selected element is inserted. Alternatively, you can drag-and-drop an element into the editor.</p>
Drag-and-drop of a function block instance from the declaration into the editor	<p>Requirement: a declaration line is selected in the declaration of the CFC.</p> <p>The instance is inserted as a function block with block name and block type and all connections.</p>
Drag-and-drop of a variable from the declaration into the editor to a block connection	<p>The variable is inserted as input or output with connection to the focused block connection.</p> <p>Tip: The cursor indicates whether you are focusing on a valid location for a variable.</p> 
Drag-and-drop of a variable from the declaration part into the editor	<p>Requirement: The respective element is selected in the declaration.</p> <ul style="list-style-type: none"> • Function block instance: a function block with the corresponding data type is created. • Declaration of VAR_INPUT or CONSTANT: an input element is inserted. • Declaration of VAR_OUTPUT: an output element is inserted. • Declaration of VAR, VAR_GLOBAL: a window opens at the insertion position where you can select whether an input or an output element is to be inserted. <p>When a variable is dragged and dropped from the declaration part onto an existing replaceable element, the existing element is replaced.</p>
Drag-and-drop of a function block or programming block from the project tree or the Library Manager into the editor	<p>A function block element with the corresponding type is inserted.</p> <ul style="list-style-type: none"> • If a function block is dragged and dropped onto an existing connecting line and both an input and an output of the function block are compatible with the data type of the line, the function block is inserted on the line. In this case, its first matching input and output are connected to the elements that were previously connected by the connecting line. • When a function block is dragged and dropped onto an existing function block, the existing function block is replaced.
Rearranging the order of inputs and outputs within a function block using drag-and-drop.	<p>Requirement: the text field of the input or output to be reordered to another position is selected.</p>



<p>[Ctrl] + click in the programming area</p>	<p>Requirement: an element is selected in the Toolbox view.</p> <p>As long as you hold down [Ctrl] , a selected element is created each time you click in the programming area.</p>
<p>[Ctrl] + right arrow</p>	<p>Requirement: in the CFC program, exactly one output pin is selected for an element.</p> <p>The selection is moved so that the input pin is selected at the end of the connecting line. If there are several pins, all of them are selected.</p> <p>Example:</p> 
<p>[Ctrl] + left arrow</p>	<p>Requirement: in the CFC program, exactly one input pin is selected for an element.</p> <p>The selection is moved so that the output pin at the beginning of the connecting line is selected. If there are several pins, all of them are selected.</p> <p>Example:</p> 

See also:

- [General functionalities in all graphical editors](#) [▶ 623]

Connect

You can insert connecting lines between element pins. Connecting lines are inserted with auto-routing so that they are automatically optimal and as short as possible. The connecting lines are checked for collisions.

Drag-and-drop from one pin to another	A connecting line is inserted between the 2 element pins.
Drag-and-drop from a pin to a function block	Drop can be performed on a pin or on the text field of a connection. For extensible operators (for example ADD) the drop can also be done within the function block. The following behavior applies: If there are still unconnected input pins, the uppermost free pin is connected. If there are no more unconnected input pins, then a new pin is automatically inserted at the bottom.
Command Connect selected pins	Requirement: you have selected several pins. The pins are marked in red.
Move inserted element so that it touches the pin of another element.	Requirement: option Enable AutoConnect is enabled. The touching pins are automatically connected to each other.
	The connection icon is located in the upper right corner in the editor. A green icon indicates collision-free connections. A red icon indicates collisions. Clicking on the icon opens a menu with commands for collision processing, for example the command Show next collision .
	Requirement: you have selected a connection and the command Connection mark . Instead of a long connecting line, a connection is represented by connection marks.

See also:

- [Command Connect Selected Pins \[► 1017\]](#)
- [Command Show Next Collision \[► 1018\]](#)
- [Command Connection Mark \[► 1022\]](#)

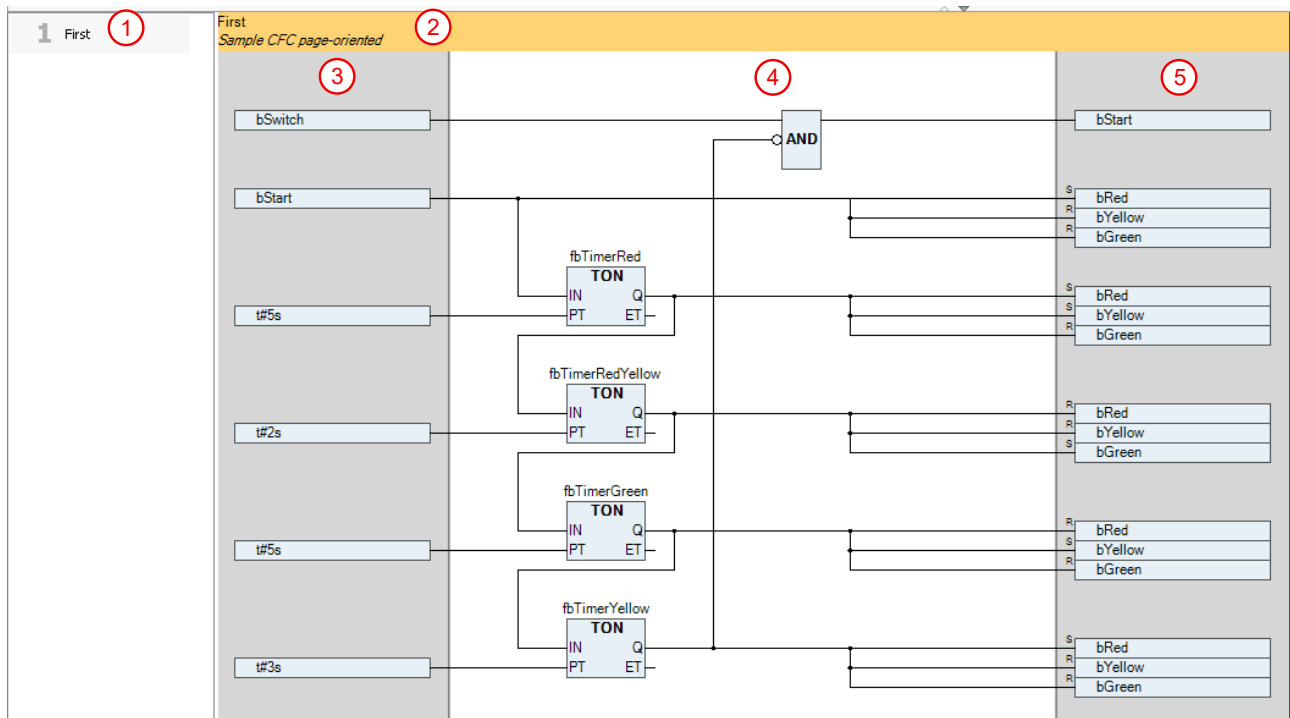
See also:

- [General functionalities in all graphical editors \[► 623\]](#)
- [Programming in CFC \[► 114\]](#)
- [Automatic execution order by data flow \[► 120\]](#)
- [Elements \[► 676\]](#)
- TC3 User Interface documentation: [CFC \[► 1011\]](#)
- TC3 User Interface documentation: [Dialog Options - CFC Editor \[► 966\]](#)

16.1.6.2 CFC Editor, Page-Oriented

You cannot convert programming blocks created in the implementation language **Continuous Function Chart (CFC) - page-oriented** into programming blocks of the implementation language **Continuous Function Chart (CFC)** and vice versa.

You can copy elements between these two editors with the **Copy** and **Paste** commands (via the clipboard), or via drag-and-drop.



1. Page navigator
2. Page header with page name and page description
3. Left margin reserved for inputs and sink connection marks
4. Program area
5. Right margin area reserved for outputs and source connection marks

Edit

You can drag an element **page** from the **toolbox** to the page navigation, then another page will be inserted.

You can select existing pages in the page navigation and multiply them with the commands **Copy** and **Paste**.

You can change the size of the page using the **Edit page size** command.

You create connections across pages using the elements **source connection mark** and **sink connection mark**. When you drag a connecting line from an input pin or an output pin to the margin, a new connection mark is automatically created. The **List components** feature provides an overview of all defined source connection marks.

Once you have selected an element in the editor, you can use the arrow keys to move the selection from one element to the next and thus navigate through the circuit. If you have selected a connection mark and still press an arrow key, the corresponding connection mark of the next/previous page will be selected.

You can transfer networks or elements from a CFC programming block to the program area of a page-oriented CFC using the commands **Copy** and **Paste**. Alternatively, you can use drag-and-drop.

Execution order

The execution order is automatically set according to the order of the pages as they are sorted in the page navigator of the editor. Within a page, a page-oriented CFC object behaves like a CFC object. You can thus switch between **Automatic data flow mode** and **Explicit execution order mode**.

Additional commands in CFC page-oriented

- [Command Edit page size \[► 1011\]](#)
- [Command Edit Worksheet \[► 1011\]](#)

See also:

- [General functionalities in all graphical editors](#) [▶ 623]
- [Programming in CFC](#) [▶ 114]
- [Automatic execution order by data flow](#) [▶ 120]
- [Elements](#) [▶ 676]
- [TC3 User Interface documentation: CFC](#) [▶ 1011]
- [TC3 User Interface documentation: Dialog Options - CFC Editor](#) [▶ 966]

16.1.6.3 CFC editor in online mode

In online mode, you can monitor controller variable values in the CFC Editor and modify them. You can also use the TwinCAT debugging functions, such as breakpoints and step-by-step execution.

Monitoring

You can observe values in the declaration part as well as in the implementation part (with inline monitoring) as usual.

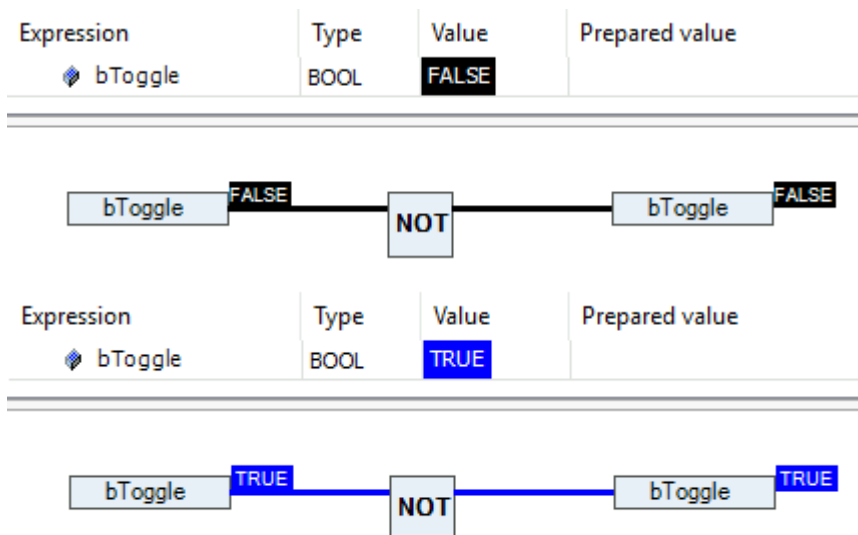
Inline monitoring for a function block is only possible if an instance of the function block is open. No values are displayed in the basic implementation view.

Monitor boolean variables

The connections between boolean variables are displayed in color according to their actual value: TRUE in blue and FALSE in black. The element pins are decorated with the actual value.

Sample

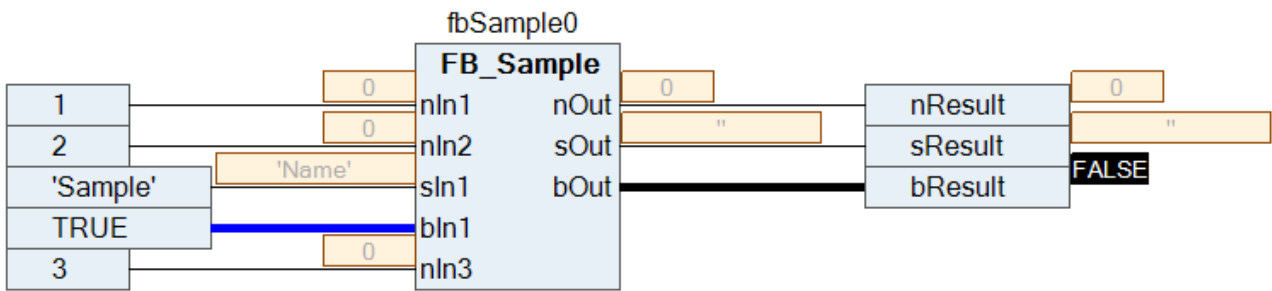
An application contains a CFC programming block. There an internal boolean variable is toggled: the variable `bToggle` changes its state from TRUE to FALSE with each bus cycle.



Monitor scalar variables

For scalar variables, the element pins are decorated with the actual values.

Sample



Forcing and Writing Variables Values

In online mode you can prepare a value for forcing or writing a mapped variable in the declaration editor.

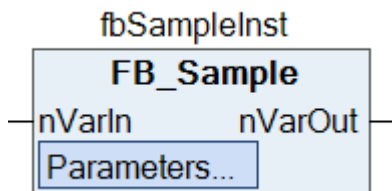
If you have enabled the option **Prepare values in implementation part** in the TwinCAT options under category **TwinCAT > PLC Environment > CFC editor**, you can also prepare values in the implementation part. To do this, open the **Prepare value** dialog by double-clicking on the monitoring box next to each element or directly on the element. For boolean variables, no dialog appears, but each mouse click on the value displayed next to the variable toggles between TRUE and FALSE.

Prepared values are displayed in angle brackets. After the writing or forcing has been executed, a red "F" appears in the monitoring box.

Change constant input parameters of function block instances

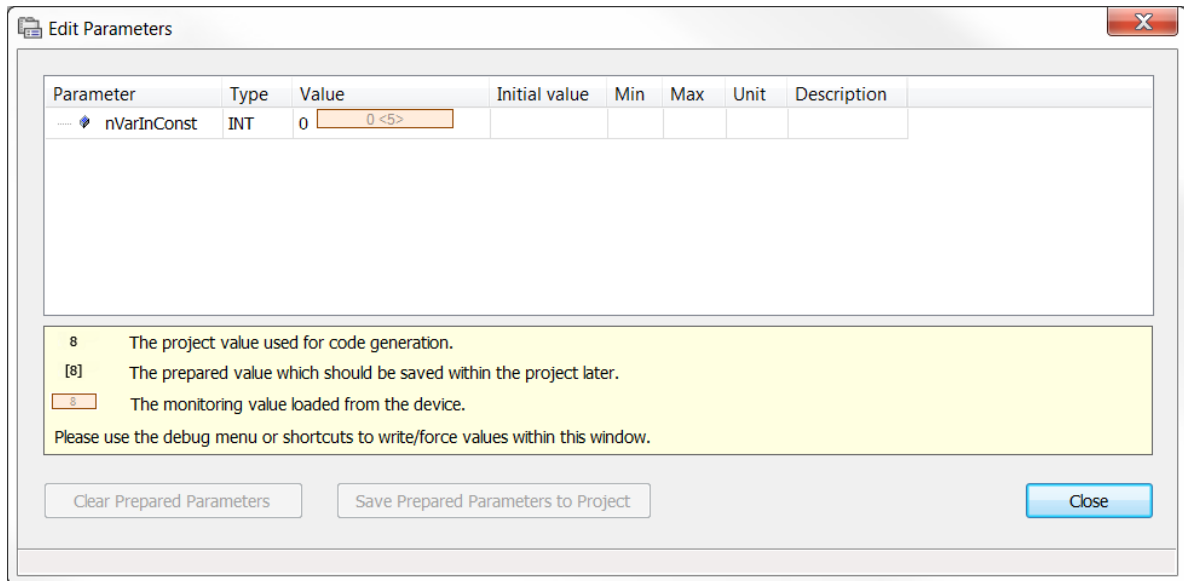
Input parameters of function block instances of type `VAR_CONSTANT` can also be changed in online mode. The parameters are adjusted accordingly. After logging out, you can apply these parameters to your project with the command **Accept prepared parameter values**.

- ✓ A CFC editor is active. A function block is instantiated, which has `VAR_INPUT CONSTANT` variables in its declaration.
- 1. Open the POU by calling the function block instance in the editor.
- 2. Log on to the controller.
- 3. Click in the **Parameter** field of one of the function block instances.



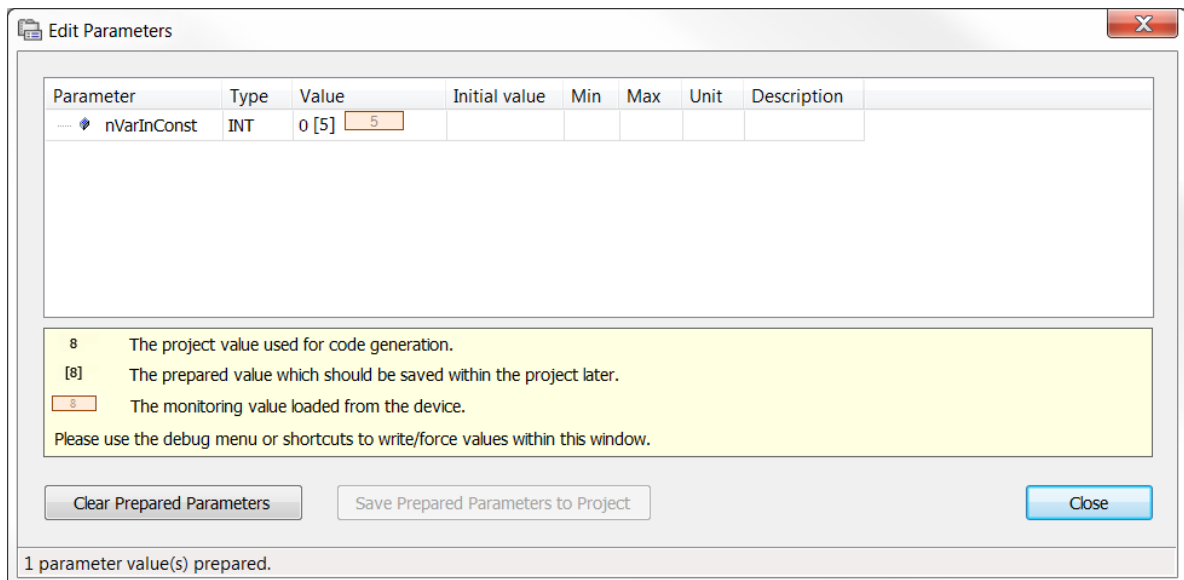
- ⇒ The **Edit Parameters** dialog opens.
- 4. In the **Value** column, click in an inline monitoring field of a parameter.
 - ⇒ The **Prepare Value** dialog opens.
- 5. Enter a new value in the field **Prepare a new value for the next write or force operation**.
- 6. Confirm the entry with **OK**.

⇒ The prepared value is displayed in angle brackets next to the current value (in the example <5>)

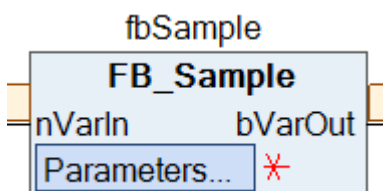


7. Write the prepared value with the command **Write values** in the **PLC** menu or in the toolbar.

⇒ The prepared value is written. The parameter is changed and is shown in square brackets.



Next to the parameter field of the function block instance, the difference of the two values is represented by a red cross



8. Close the **Edit Parameters** dialog.

9. Log out.

10. Click in the **Parameter** field of one of the function block instances or select the function block instance and use the command **Edit Parameters** in the **CFC** menu.

⇒ The **Edit Parameters** dialog opens again.

11. Select the command **Save Prepared Parameters to Project**.

⇒ The changed parameter value is transferred to the project. The asterisk next to the parameter field disappears.

See also:

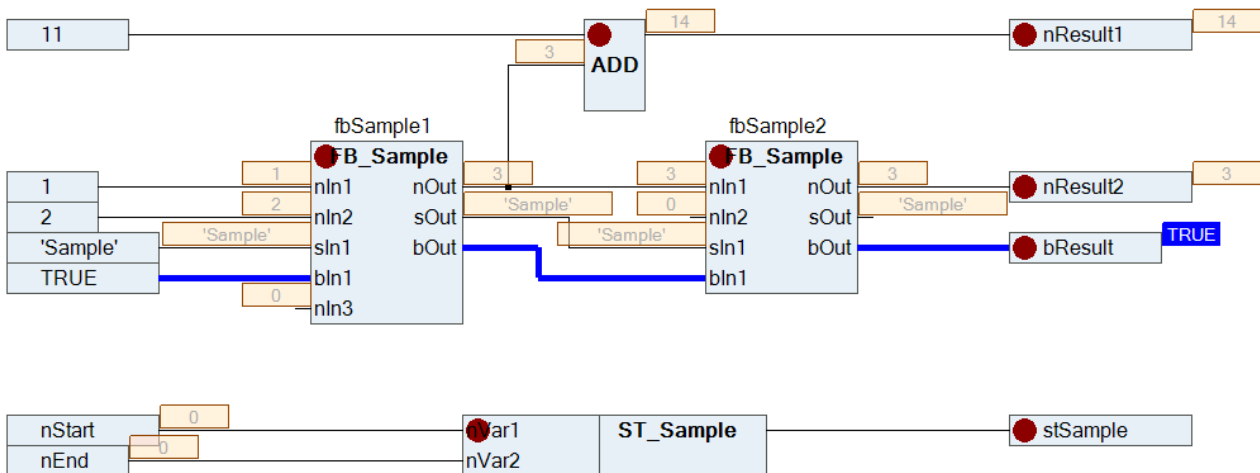
- TC3 User Interface documentation: [Command Edit Parameters \[▶ 1023\]](#)
- TC3 User Interface documentation: [Command Save prepared parameters in the project \[▶ 1024\]](#)

Breakpoint positions

Possible positions of a breakpoint

- Element **Output**
Variables are written.
- Element **Function block**
Programming blocks are called.
- Element **RETURN**
The program flow branches out.
- Element **Compositor**
Tree items are written.

Select the command **Toggle Breakpoint** from the **Debug** menu to set a breakpoint or delete an existing one. Red circles in the block diagram represent an activated breakpoint.



A breakpoint is automatically set in all methods that can be called. Thus, if a method is called which is defined via an interface, breakpoints are set in all methods of function blocks that implement this interface. This also applies in all derived function blocks that define the method.

Execute programming block step by step

You can process a programming block step by step in debug mode. A programming block that is called is internally supplemented by a RETURN at the beginning before the element with the number 0 and at the end after the last element. During step-by-step processing, these are automatically jumped to.

See also:

- [Monitoring Values \[▶ 222\]](#)
- [Forcing and Writing Variables Values \[▶ 214\]](#)
- [Use of breakpoints \[▶ 211\]](#)
- [Stepwise processing of the program \(stepping\) \[▶ 213\]](#)

Also see about this

- [Command Edit Parameters \[▶ 1023\]](#)
- [Command Save prepared parameters in the project \[▶ 1024\]](#)

16.1.6.4 Define hotkeys for the CFC editors

In the **Options** you can define individual hotkeys for many commands in the CFC editor under **Environment > Keyboard** to simplify editing:


Command	Command for hotkey
Select all	CFC.SelectAll
Insert elements:	
Insert box. To select the box, the dialog Input Assistant opens.	CFC.InsertBoxFromInputAssistant
Insert Empty Box.	CFC.InsertBox
Insert Box with EN/ENO. To select the box, the dialog Input Assistant opens.	CFC.InsertBoxWithENENOfromInputAssistant
Insert Input. Inserts an input element.	CFC.InsertInput
Insert Output. Inserts an output element.	CFC.InsertOutput
Insert Jump.	CFC.InsertJump
Edit elements that have already been inserted:	
Negate	CFC.Negate
Switch between Set, Reset, REF and None.	CFC.SwitchSRRefNone
Reset Pins.	CFC.ResetPins

See also

- TC3 User Interface documentation: [CFC \[► 1011\]](#)
- TC3 User Interface documentation: [Customizing Keyboard Shortcuts](#)

16.1.6.5 Elements

16.1.6.5.1 CFC element Page


Symbol: 

The element inserts a new page into the editor. It is only available in the page-oriented CFC editor. The page number is automatically assigned based on its position. You can enter the page name and description in the orange header. Use the **Edit page size** command to adjust the page size.

See also:

- TC3 User Interface documentation: [Command Edit page size \[► 1011\]](#)

16.1.6.5.2 CFC element Control point


Symbol: 

You can use a control point to fix points of a connection before you adjust the line routing. To do this, drag the element to the desired position on a connecting line. Connection lines with control points are no longer routed automatically.

See also:

- [Programming in CFC \[► 114\]](#)
- TC3 User Interface documentation: [Command Create Control Point \[► 1021\]](#)
- TC3 User Interface documentation: [Command Remove Control Point \[► 1021\]](#)


16.1.6.5.3 CFC element Input

Symbol: 

By default, TwinCAT adds an input element with the text **???**. You can edit this field directly by clicking on it and enter a constant value or a variable name. Alternatively, you can open the input assistant by clicking

 to select a variable.


16.1.6.5.4 CFC element Output


Symbol: 

By default, TwinCAT adds an output element with the text **???**. You can edit this field directly by clicking on it and enter a constant value or a variable name. Alternatively, you can open the input assistant by clicking

 to select a variable.

16.1.6.5.5 CFC element function block

Symbol: 

You can use the element to insert an operator, a function, a function block or a program. By default, TwinCAT adds the element with the name **???**. You can edit this field directly by clicking on it and enter a function block name. Alternatively, you can open the input assistant by clicking  and select a function block.

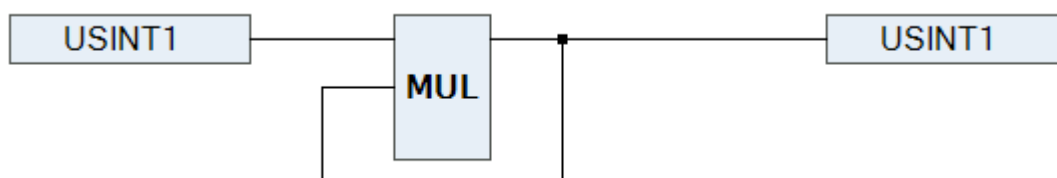
In the case of a function block, TwinCAT also displays an input field above the function block symbol (**???**). You have to replace this name with the name of the function block instance. If you instantiate a function block with constant input parameters, the function block element displays the field **Parameter...** in the lower left corner. You can edit the parameters by clicking on this field.

To replace an existing box, simply replace the currently inserted identifier with the desired new name. Note that TwinCAT adapts the number of input and output pins according to the definition of the POU and therefore deletes any existing assignments.

i As in the CFC feedbacks are allowed, implicit variables with data type of the input variables are generated at the output of a function block (in the sample `temp_USINT`). If the result of the operation of a function block is a value that exceeds the number range of the data type of the input variable, the overflow is written to the implicit variable. The actual output variable receives the value of the implicit variable, i.e. the overflow and not the actual result of the operation (see sample).

Sample

Implicitly generated variables at the function block output:



Implicitly generated code:


```
temp_USINT := USINT1 * temp_USINT;
UDINT1    := temp_USINT;
```

See also:

- TC3 User Interface documentation: [Command Edit Parameters](#) |▶ 1023]

16.1.6.5.6 CFC element Jump


Symbol: 

You can use the element to specify a position where program processing should continue. You have to define this target position with a marker. To do so, enter the name of the marker in the input field ????. If you have already inserted the corresponding marker, you can select it using the input wizard ().

See also:

- [CFC element Mark \[▶ 678\]](#)

16.1.6.5.7 CFC element Mark

Symbol: 


A mark defines a position to which the program jumps during processing using a jump element.

In online mode TwinCAT automatically inserts a RETURN mark at the end of a CFC function block.

See also:

- [CFC element Jump \[▶ 678\]](#)


16.1.6.5.8 CFC element Return

Symbol: 

Use the element to exit the function block.

Note that in online mode a return element is automatically inserted before the first line and after the last element in the CFC editor. With step-by-step processing, TwinCAT automatically jumps to the return element before the function block is exited.

16.1.6.5.9 CFC element Composer

Symbol: 


The composer element is used to handle structure components. The individual components of a structure are made available as input. To this end, you have to assign the same name to the composer element as the corresponding structure (replace the ???).

The composer element is the counterpart to the selector element.

See also:

- [CFC element Selector \[▶ 678\]](#)

16.1.6.5.10 CFC element Selector

Symbol: 


The selector element is used to handle structure components. The individual components of a structure are made available as output. To this end, you have to assign the same name to the selector element as the corresponding structure (replace the ???).

The selector element is the counterpart to the composer element.

See also:

- [CFC element Composer](#) [▶ 678]

16.1.6.5.11 CFC element Comment

Symbol: 

You can use this element to enter a comment in the CFC editor. Replace the placeholder text in the element with the comment text. To insert a line break, use the key combination **[Ctrl] + [Enter]**.

16.1.6.5.12 CFC element Connection mark source/target

Symbol:  , 

You can use connection marks instead of a connection line between elements. This helps to make complex diagrams easier to read.

For a valid connection, you have to link a **Connection mark - source** element to the output of an element and a **Connection mark - target** element to the input of another element. Both marks have to have the same name. The name is not case-sensitive.

Notes on naming:

- The default name for connection marks is C-<nr>. <nr> is a consecutive number, starting with 1.
- You can change the default name. Ensure that the source and the destination mark have the same name.
- If you change the name of the source mark, the destination name is automatically renamed.
- If you change the name of the target mark, the source name is retained.



Note the command to automatically convert an existing connection.

See also:

- TC3 User Interface documentation: [Command Connection Mark](#) [▶ 1022]

16.1.6.5.13 CFC element function block input

Symbol: 


Depending on the function block type, you can add further inputs to an inserted function block element. To do this, select the function block element and drag the function block input element to the function block body.

You can drag an input or output connection to a different position on the function block while pressing the **[Ctrl]** key.

See also:

- [CFC element function block output](#) [▶ 679]

16.1.6.5.14 CFC element function block output

Symbol: 

Depending on the function block type, you can add further outputs to an inserted function block element. To do this, select the function block element and drag the function block output element to the function block body.

You can drag an input or output connection to a different position on the function block while pressing the **[Ctrl]** key.

See also:

- [CFC element function block input \[▶ 679\]](#)

16.1.7 Ladder Diagram (LD) (Beta)

16.1.7.1 Ladder editor



The new Ladder editor is available in a beta version from TC3.1 Build 4026.


The Ladder editor is a network-based editor for the IEC programming language Ladder Diagram (LD). In the implementation part, you program a "circuit diagram" in one or more network elements. To do this, you can insert the required elements from the toolbox or by means of menu commands, link them and provide them with input and output variables and modifiers.

Each network can be given with a title, a comment and/or a jump label. You can comment out networks.

Inserting and replacing elements is done by dragging an element with the mouse from the toolbox to one of the offered insertion positions.

The insertion positions are displayed with the following symbols while you drag the element over the implementation part:

- Square with a gray background inside an existing element symbol
- Rhombus on a connecting line
- Triangle pointing up or down for insertion above or below

A possible position "lights up", the mouse pointer gets a plus sign , and when the mouse button is released, the element is inserted.

Currently selected areas in the editor are highlighted in red and outlined.

Suitable commands for modifying (Edge Detection, Set/Reset, EN/ENO), deleting, refactoring, searching and so on are each located in the context menu.

In [online mode \[▶ 680\]](#), monitoring and troubleshooting are possible using breakpoints and writing and forcing values.

The representation in the editor is defined in the TwinCAT options, category [Ladder \[▶ 983\]](#) editor. This concerns, for example, the display of comments, addresses or whether networks are provided with line breaks.

Programs created in the LD/FBD editor can be read into the Ladder editor and edited further.

See also:

- [Programming in the Ladder editor \[▶ 128\]](#)
- User interface documentation: [Ladder editor \[▶ 1038\]](#)
- User interface documentation: [Dialog: Options – Ladder editor \[▶ 983\]](#)

16.1.7.2 Ladder editor in online mode

In online mode, the editor provides value monitoring and supports the writing and forcing of current values. You can set breakpoints, and the color-coded representation of the connections allows for a kind of flow control.

Monitoring

In online mode, its actual value is displayed next to each variable in the editor. Constant variables get a green C symbol. The representation of the values is defined in the TwinCAT options, category Ladder editor [[983](#)].

Writing/forcing of values

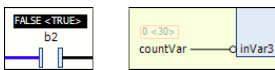
If a variable is currently forced, this is indicated by **F** directly in front of the forced value. If a value is prepared for writing or forcing, this value is displayed directly after the actual value in angle brackets: *<value>*.

Examples:

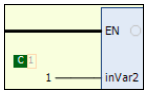
Forced variable:



Prepared value:



Constant value:



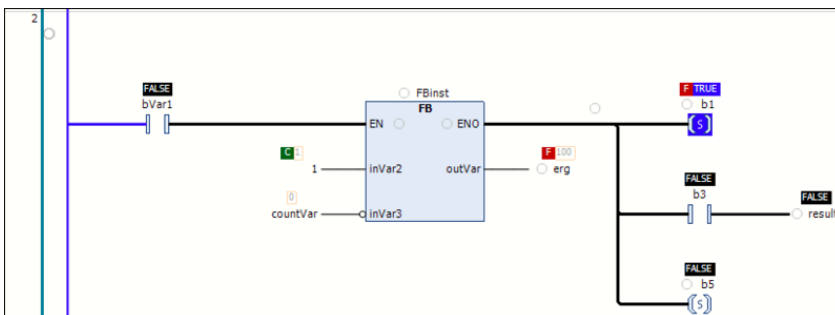
Color-coded representation of connections

In the online view of a Ladder diagram, the connecting lines are displayed in color: connections with the value *TRUE* are indicated by a thick blue line and connections with the value *FALSE* by a thick black line. Connections of unknown or analog value are displayed normally (thin black line).



The value of the connections is not read from the monitored variables, but calculated in the programming system. This is not a real flow control.

Example: Connecting lines and breakpoint positions



Breakpoints

Breakpoints are basically possible at positions where the values of variables can change (statement), where the program branches, or where another programming block is called.

In the editor, possible breakpoint positions are indicated by an empty gray circle. Set breakpoints are displayed as a solid red circle. See the figure above: "Example: Connecting lines and breakpoint positions".

Possible breakpoint positions:

- On a callable box (function block, function, program, action, method). Not possible with operator boxes (example: *ADD*, *DIV*)
- On assignments

- Before parallel branches
- At the end of the box at the position of the return to the calling box. In online mode, an empty network automatically appears at this point, which is marked with **RET** instead of a network number.
- On **EN** input and **ENO** output of a box
- On the entire network. Indicates only that a breakpoint is set in the network. No breakpoint can be set on the entire network.

i Breakpoints in methods: TwinCAT automatically sets a breakpoint in all methods that can be called. This means that if a method managed by an interface is called, breakpoints are set in all methods that occur in function blocks that implement this interface, as well as in all derived function blocks that use the method. If a method is called via a pointer to a function block, TwinCAT sets the breakpoints in the method of the function block and in all derived function blocks that use the method.

16.1.7.3 Branch

A line branch splits the processing line into 2 or more branches, which are executed one after the other from top to bottom. You can further branch each branch, which results in multiple branches being created within a network.

Each branch gets a marker symbol (small rectangle) at the branching point, which you can select to perform further actions/commands for the subnet.

For more information, see [Programming in the Ladder editor \[▶ 128\]](#)

16.1.7.4 Navigation in the Ladder editor

You can use the keys and key combinations as described below to switch between cursor positions in the editor. Switching also works across networks.

→ ←	Change to the adjacent cursor position, along the signal flow, i.e. from left to right and vice versa.
↑ ↓	Change to the next cursor position above or below the current position, if this adjacent position belongs to the same logical group. A logical group is formed by all the connections of a box, for example. If such a logical group does not exist: Move to the first cursor position in the next upper or lower adjacent element. In the case of elements connected in parallel, navigation takes place along the first branch.
Not currently implemented. Ctrl + Pos1	Change to the first network; this is selected.
Ctrl + End	Change to the last network; this is selected.
Page Up ↑	Scroll up 1 page. The top network on this page is selected.
Page Down ↓	Scroll down 1 page. The bottom network on this page is selected.

16.1.7.5 Elements

The elements that are available in the Ladder editor are described in the help pages for the menu commands that are available for inserting the individual elements.

16.2 Variables

The scope of a variable defines how and where you can use the variable. You specify the scope of validity in the variable declaration.

16.2.1 Local Variables - VAR

Local variables are declared in the declaration part of programming objects between the keywords VAR and END_VAR.

You can extend local variables with an attribute keyword.

You can access local variables for reading from outside the programming objects via the instance path. Access for writing from outside the programming object is not possible; this will be displayed by the compiler as an error.

In order to maintain the intended data encapsulation, it is strongly recommended not to access local variables of a POU from outside the POU – neither in read mode nor in write mode. (Other high-level language compilers also output read access operations to local variables as errors.) Furthermore, with library function blocks it cannot be guaranteed that the local variables of a function block will remain unchanged during subsequent updates. This means that it is possible that the application project can no longer be compiled correctly after the library update.

Also observe here rule SA0102 from the Static Analysis, which determines access to local variables for reading from outside.

Sample:

```
VAR
  nVar1 : INT;
END_VAR
```

See also:

- Documentation TE1200 | PLC Static Analysis: [Configuration > Rules](#)
- [Variable types - attribute keywords \[► 695\]](#)

16.2.2 Input Variables - VAR_INPUT

Input variables are input variables for a function block.

VAR_INPUT variables are declared in the declaration part of programming objects between the keywords VAR_INPUT and END_VAR.

You can extend input variables with an attribute keyword.

Example:

```
VAR_INPUT
  nIn1 : INT; //1st input variable
END_VAR
```

See also:

- [Variable types - attribute keywords \[► 695\]](#)

16.2.3 Output Variables - VAR_OUTPUT

Output variables are output variables of a function block.

VAR_OUTPUT variables are declared in the declaration part of programming objects between the keywords VAR_OUTPUT and END_VAR. TwinCAT returns the values of these variables to the calling function block. There you can query the values and continue to use them.

You can extend output variables with an attribute keyword.

Example:

```
VAR_OUTPUT
  nOut1 : INT; //1st output variable
END_VAR
```

Output variables in functions and methods

According to the IEC 61131-3 standard, functions (and methods) can have additional outputs. You have to assign the additional outputs when you call the function, as shown in the following example.

Example:

```
F_Fun(nIn1 := 1, nIn2 := 2, nOut1 => nLoc1, nOut2 => nLoc2);
```

See also:

- [Variable types - attribute keywords \[► 695\]](#)

16.2.4 Input/Output Variables - VAR_IN_OUT, VAR_IN_OUT CONSTANT

A VAR_IN_OUT variable is an input and output variable that is part of a function block interface and serves as a formal pass-through parameter.

The VAR_IN_OUT variables of a function block must be assigned when the function block is called.

Syntax:

```
<keyword> <POU name>
VAR_IN_OUT
  <variable name> : <data type> ( := <initialization value> )? ;
END_VAR
<keyword> : FUNCTION | FUNCTION_BLOCK | METHOD | PRG
```

An input and output variable can be declared in the programming blocks PRG, FUNCTION_BLOCK, METHOD or FUNCTION in the declaration section VAR_IN_OUT. A constant of the declared data type can optionally be assigned as the initialization value. The VAR_IN_OUT variable can be read and written.

Use:

- **Call:** When the programming block is called, the formal VAR_IN_OUT variable receives the actual variable (referred to as pass-through variable) as an argument. No copy is created at runtime during the parameter passing, but the formal variable receives a reference to the actual variable transferred from outside. The internal value of the referential variables is a memory address to the actual value (transfer as pointer, call by reference). It is not possible to directly specify a constant (literal), a constant variable or a bit variable as an argument.
- **Read/write access within the programming block:** Any write access to the variable within the programming block has an effect on the transferred variable. When the programming block is exited, any changes made are thus retained. This means that a programming block uses its VAR_IN_OUT variables in the same way as the calling programming block uses its variables. Read access is always allowed.
- **Read/write access from outside:** VAR_IN_OUT variables cannot be read or written directly from outside via <function block instance name>.<variable name>. This only works for VAR_INPUT and VAR_OUTPUT variables.
- **Transferring string variables:** If a string variable is transferred as an argument, the actual variable and the formal variable should have the same length. Otherwise the transferred string can be manipulated unintentionally. This problem does not occur with VAR_OUTPUT CONSTANT parameters.
- **Transferring bit variables:** A bit variable cannot be transferred directly to a VAR_IN_OUT variable but requires an intermediate variable.
- **Transfer of properties:** not allowed.

● Transferring strings to VAR_IN_OUT CONSTANT

I If a character string is transferred as a variable or as a constant to a formal VAR_IN_OUT CONSTANT variable, the string length can vary. At the most, however, the string length of the VAR_IN_OUT CONSTANT variable will be processed. Further information can be found at the bottom of this page.

i Transfer of properties not possible

No property can be transferred to a VAR_IN_OUT or VAR_IN_OUT CONSTANT variable.

Exception: An assignment to VAR_IN_OUT (CONSTANT) is possible if the property has the return type REFERENCE. Note, however, that the property may only return an address via REFERENCE that is valid beyond the property call. This would not be the case, for example, if the property returned the reference to a temporary variable.

Sample:

Function block FB_Sample

```
FUNCTION_BLOCK FB_Sample
VAR_IN_OUT
  bInOut    : BOOL;
END_VAR
```

Program MAIN:

```
VAR
  bTest      : BOOL;
  fbSample   : FB_Sample;
END_VAR

fbSample(bInOut := bTest); // OK
fbSample();              // NOK: not possible as the VAR_IN_OUT variable is not assigned in this
                          // FB call
fbSample.bInOut := bTest; // NOK: direct access to VAR_IN_OUT variable from the outside not
                          // possible
```

Sample of a workaround for assigning a bit variable to a VAR_IN_OUT input:

Declaration of the bit variables (bBit0):

```
VAR_GLOBAL
  bBit0 AT %MX0.1 : BOOL;
  bTemp          : BOOL;
END_VAR
```

Function block with VAR_IN_OUT input bInOut:

```
FUNCTION_BLOCK FB_Test
VAR_INPUT
  bIn    : BOOL;
END_VAR
VAR_IN_OUT
  bInOut : BOOL;
END_VAR

IF bIn THEN
  bInOut := TRUE;
END_IF
```

Program that calls the function block. Direct assignment of the bit variable to the VAR_IN_OUT input (error) and assignment using an intermediate variable (workaround):

```
PROGRAM MAIN
VAR
  bIn      : BOOL;
  fbTest1  : FB_Test;
  fbTest2  : FB_Test;
END_VAR

// Error C0201: Type 'BIT' doesn't correspond to the type 'BOOL' of VAR_IN_OUT 'bInOut'
fbTest1(bIn := bIn, bInOut := bBit0);

// Workaround
//bTemp := bBit0;
//fbTest2(bIn := bIn, bInOut := bTemp);
//bBit0 := bTemp;
```

Transfer variable VAR_IN_OUT CONSTANT

A VAR_IN_OUT CONSTANT variable serves as a constant transfer parameter that can be read but not written to.

Syntax:

```
<keyword> <POU name>
VAR_IN_OUT CONSTANT
  <variable name> : <data type>; // formal parameter
END_VAR
<keyword> : FUNCTION | FUNCTION_BLOCK | METHOD | PRG
```

VAR_IN_OUT CONSTANT variables are declared without assigning an initialization value.

Use:

- All data types are allowed.
- Write access to the VAR_IN_OUT CONSTANT variable is not permitted.
- Transfer of properties is not permitted.
- If the VAR_IN_OUT CONSTANT variable is of type STRING/WSTRING, a variable, a constant variable, or a constant (literal) can be transferred when the programming block is called. There are no restrictions on the string length of the transferred variable/constant, and the length does not depend on the string length of the VAR_IN_OUT CONSTANT variable. Note that at the most, the string length of the VAR_IN_OUT CONSTANT variable will be processed.
- If the VAR_IN_OUT CONSTANT variable is not of type STRING/WSTRING, a variable can be transferred when the programming block is called. A constant variable can be transferred if the compiler option **Replace constants** is disabled in the **Compile** category of the PLC project properties.



If the compiler option **Replace constants** is enabled in the **Compile** category of the PLC project properties, the parameter transfer of a constant with a basic data type other than STRING or a constant variable with a basic data type other than STRING generates a compiler error.

Sample:

In the code, strings are transferred to the F_Manipulate function via various VAR_IN_OUT variables. A compiler error is issued if a literal is transferred to a VAR_IN_OUT variable. Correct code is generated when a literal is transferred to a VAR_IN_OUT CONSTANT variable and when string variables are transferred.

It should also be noted that it is not possible to transfer a STRING variable that is too short to a VAR_IN_OUT variable (compiler error), although this is possible when transferring to a VAR_IN_OUT CONSTANT variable.

Function F_Manipulate:

```
FUNCTION F_Manipulate : BOOL
VAR_IN_OUT
  sReadWrite   : STRING(16); (* Can be read or written here in POU *)
  nReadWrite   : DWORD;     (* Can be read or written here in POU *)
END_VAR
VAR_IN_OUT CONSTANT
  cReadOnly    : STRING(16); (* Constant string variable can only be read here in POU *)
END_VAR
```

```
sReadWrite := 'String_from_POU';
nReadWrite := STRING_TO_DWORD(cReadOnly);
```

Program MAIN:

```
PROGRAM MAIN
VAR
  sVar10 : STRING(10) := '1234567890';
  sVar16 : STRING(16) := '1234567890123456';
  sVar20 : STRING(20) := '12345678901234567890';
  nVar   : DWORD;
END_VAR

// The following line of code causes the compiler error:
// VAR_IN_OUT parameter 'sReadWrite' of 'F_Manipulate' needs variable with write access as input.
F_Manipulate(sReadWrite := '1234567890123456', cReadOnly := '1234567890123456', nReadWrite := nVar);

// The following line of code causes the compiler error:
// String variable 'sVar10' too short for VAR_IN_OUT parameter 'sReadWrite' of 'F_Manipulate'
F_Manipulate(sReadWrite := sVar10, cReadOnly := sVar10, nReadWrite := nVar);

// Correct code
F_Manipulate(sReadWrite := sVar16, cReadOnly := '1234567890', nReadWrite := nVar);
F_Manipulate(sReadWrite := sVar16, cReadOnly := '1234567890123456', nReadWrite := nVar);
```

```
F_Manipulate(sReadWrite := sVar16, cReadOnly := '12345678901234567890', nReadWrite := nVar);
F_Manipulate(sReadWrite := sVar16, cReadOnly := sVar10, nReadWrite := nVar);
F_Manipulate(sReadWrite := sVar16, cReadOnly := sVar16, nReadWrite := nVar);
F_Manipulate(sReadWrite := sVar20, cReadOnly := sVar20, nReadWrite := nVar);
```

16.2.5 Global Variables - VAR_GLOBAL

Global variables are "normal" variables, constants, or retentive variables that are known throughout the project.

Global variables are declared in global variable lists between the keywords VAR_GLOBAL and END_VAR.

The system detects a global variable if the variable name is prefixed with a dot, for example ".nGlobVar1".



A variable declared locally in a function block, which has the same name as a global variable, has priority within the function block.



Global variables are always initialized before the local variables of POU's.

Example:

```
VAR_GLOBAL
    nVarGlob1 : INT;
END_VAR
```

See also:

- [Object Global Variable List \[▶ 72\]](#)
- [Global namespace \[▶ 715\]](#)

16.2.6 Temporary Variable - VAR_TEMP

This functionality is an extension with regard to the IEC 61131-3 standard.

Temporary variables are declared locally between the keywords VAR_TEMP and END_VAR.

VAR_TEMP declarations are only possible in programs and function blocks.

TwinCAT reinitializes temporary variables each time the function block is called.

The application can only access temporary variables in the implementation part of a program or function block.

Example:

```
VAR_TEMP
    nVarTmp1 : INT; //1st temporary variable
END_VAR
```

16.2.7 Static Variables - VAR_STAT

This functionality is an extension with regard to the IEC 61131-3 standard.

You declare static variables locally between the keywords VAR_STAT and END_VAR. TwinCAT initializes the static variables when the respective function block is first called.

You can access static variables only within the namespace where the variables are declared (as is the case with static variables in C). However, static variables retain their value when the application exits the function block. You can use static variables, as counters for function calls, for example.

You can extend static variables with an attribute keyword.

Static variables only exist once. This also applies to static variables of a function block or a function block method, even if the function block is instantiated multiple times.

Example:

```
VAR_STAT
  nVarStat1 : INT;
END_VAR
```

See also:

- [Variable types - attribute keywords \[► 695\]](#)

16.2.8 External Variables - VAR_EXTERNAL

External variables are global variables that are "imported" into a function block.

You can declare the variables between the keywords VAR_EXTERNAL and END_VAR. If the global variable does not exist, an error message is issued.



In TwinCAT 3 PLC it is not necessary for variables to be declared as external to use them in a POU. The keyword exists in order to maintain compatibility with IEC 61131-3.

Syntax:

```
<POU keyword> <POU name>
VAR_EXTERNAL
  <variable name> : <data type>;
END_VAR
```

Initialization is not allowed.



Make sure that you address the allocated variables (with AT %I or AT %Q) only in the global variable list. Additional addressing of the local variables instances would result in duplications in the process image.

Sample:

```
FUNCTION_BLOCK FB_Sample
VAR_EXTERNAL
  nVarExt1 : INT;    // 1st external variable
END_VAR
```

See also:

- [Object Global Variable List \[► 72\]](#)

16.2.9 Instance Variables - VAR_INST

TwinCAT creates a VAR_INST variable of a method not on the method stack like VAR variables but on the stack of the function block instance. This means that the VAR_INST variable behaves like other variables of the function block instance and is not reinitialized each time the method is called.

VAR_INST variables are only allowed in methods of a function block, and access to such a variable is only available within the method. You can monitor the variable values of instance variables in the declaration part of the method.

Instance variables cannot be extended with an attribute keyword.

Example:

```
METHOD MethLast : INT
VAR_INPUT
  nVar : INT;
END_VAR
VAR_INST
  nLast : INT := 0;
END_VAR
```

```
MethLast := nLast;
nLast    := nVar;
```

See also:

- [Object Method \[► 90\]](#)

16.2.10 Constant variables - CONSTANT

Constant variables are declared in the global variable lists or in the declaration part of programming objects. In implementations, constant variables can be accessed for reading via the instance path, but not for writing.

Syntax

```
<scope> CONSTANT
  <identifier> : <data type> := <initial value> ;
END_VAR

<scope> : VAR | VAR_INPUT | VAR_STAT | VAR_GLOBAL
<data type>: <elementary data type> | <user defined data type> | <function block>
<initial value> : <literal value> | <identifier> | <expression>
```

Always assign an initialization value when declaring a constant variable. After that, the constant can no longer be written.

Sample:**Declaration:**

```
VAR CONSTANT
  cTaxFactor : REAL := 1.19;
END_VAR
```

Call:

```
rPrice := rValue * cTaxFactor;
```

You can only access constant variables in a read-only implementation. Constant variables are located to the right of the allocation operator.

See also:

- [Input/Output Variables - VAR IN OUT, VAR IN OUT CONSTANT \[► 684\]](#)
- [Operands \[► 744\]](#)

16.2.11 Generic constant variables - VAR_GENERIC CONSTANT

A generic constant is a variable in the VAR_GENERIC CONSTANT declaration section of a function block that is not defined until the function block instance is declared.



Available from TC3.1 Build 4026

Syntax**Declaration of the function block:**

```
FUNCTION_BLOCK <function block name>
VAR_GENERIC CONSTANT
  <generic constant name> : <integer data type> := <initial value>; //Initialwert wird
überschrieben
END_VAR
```

The generic constant (integer data type) can be used within a function block as usual. For example, you can use the constant to define the size of arrays or the length of strings. The initial value is only needed for compile checks. At runtime, the value is overwritten.

Declaration of the function block instance:

```
PROGRAM MAIN
VAR
  <fb instance name> : <function block name> < <literal> >;
  <fb instance name> : <function block name> <( <expression> )>;
END_VAR
```

When the function block instances are declared, the constant is assigned a specific value that is valid only for this instance. For this purpose, the value (<literal>) is appended in angle brackets to the function block acting as data type.

Alternatively, an expression (<expression>) can be appended. However, this must be enclosed in round brackets, since it is allowed to use symbols like < or > in an expression. Otherwise, the uniqueness of the code is no longer guaranteed.

Sample

Generic function block with parameterizable array variable

The following code shows how to define a function block that can process arrays of any length. The function block has an array with a generic but constant length. By "constant" is meant that although each function block instance varies in its array length, it is constant during the lifetime of the object.

Such a construct is beneficial, for example, to a library programmer who wishes to implement a generic library function block.

```
FUNCTION_BLOCK FB_Sample
VAR_GENERIC CONSTANT
  nMaxLen : UDINT := 1;
END_VAR
VAR
  aSample : ARRAY[0..nMaxLen-1] OF BYTE
END_VAR

PROGRAM MAIN
VAR CONSTANT
  cConst : DINT := 10;
END_VAR
VAR
  fbSample1 : FB_Sample<100>;
  fbSample2 : FB_Sample<(2*cConst)>;
  aSample : ARRAY[0..5] OF FB_Sample<10>;
END_VAR
```

Inheritance

A function block with generic constants can also use the inheritance constructs EXTENDS and IMPLEMENTS.

When implementing, make sure that the declaration of the generic constants is inserted first, followed by EXTENDS and IMPLEMENTS. The reason for this is that generic constants can also be used with base classes.

Sample

```
INTERFACE I_Sample
FUNCTION_BLOCK FB_Sample
VAR_GENERIC CONSTANT
  nMaxLen : UDINT := 1;
END_VAR
IMPLEMENTS I_Sample
VAR
  aSample : ARRAY[0..nMaxLen-1] OF BYTE
END_VAR

FUNCTION_BLOCK FB_Sub_1 EXTENDS FB_Sample<100>

FUNCTION_BLOCK FB_Sub_2
VAR_GENERIC CONSTANT
  nMaxLen2 : UDINT := 1;
END_VAR
EXTENDS FB_Sample<nMaxLen2>

PROGRAM MAIN
VAR
  fbSample1 : FB_Sample<10>;
```

```
fbSub1 : FB_Sub_1;
fbSub2 : FB_Sub_2<20>;
END_VAR
```

16.2.12 Remanent Variables - PERSISTENT, RETAIN

Remanent variables can retain their values beyond the usual program runtime. Remanent variables can be declared as RETAIN variables or even more strictly as PERSISTENT variables in the PLC project.

A prerequisite for the full functionality of RETAIN variables is a corresponding memory area on the controller (NovRam). Persistent variables are written only when TwinCAT shuts down. This will require usually a corresponding UPS. Exception: Persistent variables can also be written with the FB_WritePersistentData function block.

If the corresponding memory area does not exist, the values of RETAIN and PERSISTENT variables are lost during a power failure.



The AT declaration must not be used in combination with VAR RETAIN or VAR PERSISTENT.

Persistent Variables

You can declare persistent variables by adding the keyword PERSISTENT after the keyword for the variable type (VAR, VAR_GLOBAL, etc.) in the declaration part of programming objects.

PERSISTENT variables retain their value after an uncontrolled termination, a Reset cold or a new download of the PLC project.

When the program restarts, the system continues to operate with the stored values. In this case TwinCAT reinitializes "normal" variables with their explicitly specified initial values or with the default initializations. In other words, TwinCAT only reinitializes PERSISTENT variables during a Reset origin.

An application example for persistent variables is an operating hour counter, which is to continue counting after a power failure and when the PLC project is downloaded again.

Overview table showing the behavior of PERSISTENT variables

After online command	VAR PERSISTENT
Reset cold	Values are retained
Reset origin	Values are reinitialized
Download	Values are retained
Online Change	Values are retained

Sample:

Persistent variable list:

```
VAR_GLOBAL PERSISTENT
  nVarPers1 : DINT; (* 1. Persistent variable *)
  bVarPers2 : BOOL; (* 2. Persistent variable *)
END_VAR
```



- Avoid using the POINTER TO data type in persistent variable lists, since the address values may change when the PLC project is downloaded again! TwinCAT issues corresponding compiler warnings.
- Declaring a local variable as a PERSISTENT in a function has no effect. Data persistence cannot be used in this way.
- The behavior during a **cold reset** can be influenced with the pragma '[TcInitOnReset \[► 828\]](#)'

RETAIN variables

You can declare RETAIN variables by adding the keyword RETAIN after the keyword for the variable type (VAR, VAR_GLOBAL, etc.) in the declaration part of programming objects.

Variables declared as RETAIN are target system-dependent, but typically managed in a separate memory area that must be protected against power failure. The so-called Retain handler ensures that the RETAIN variables are written at the end of a PLC cycle and only in the corresponding area of the NovRam. The handling of the retain handler is described in chapter "[Retain data](#)" of the C/C++ documentation.

RETAIN variables retain their value after an uncontrolled termination (power failure). When the program restarts, the system continues to operate with the stored values. In this case TwinCAT reinitializes "normal" variables with their explicitly specified initial values or with the default initializations. TwinCAT reinitializes RETAIN variables at a reset origin.

One possible application is a part counter in a production plant, which should continue to count after a power failure.

Overview table showing the behavior of RETAIN variables

After online command	VAR RETAIN
Reset cold	Values are retained
Reset origin	Values are reinitialized
Download	Values are retained

Samples:

In a POU:

```
VAR RETAIN
  nRem1 : INT;
END_VAR
```

In a GVL:

```
VAR_GLOBAL RETAIN
  nVarRem1 : INT;
END_VAR
```



- If you declare a local variable as a RETAIN in a program or a function block, TwinCAT stores this specific variable in the retain area (like a global RETAIN variable).
- If you declare a local variable in a function as RETAIN, this has no effect. TwinCAT does not store the variable in the retain area.

Complete overview table

The degree of retention of RETAIN variables is automatically included in that of PERSISTENT variables.

After online command	VAR	VAR RETAIN	VAR PERSISTENT
Reset cold	Values are reinitialized	Values are retained	Values are retained
Reset origin	Values are reinitialized	Values are reinitialized	Values are reinitialized
Download	Values are reinitialized	Values are retained	Values are retained
Online Change	Values are retained	Values are retained	Values are retained

See also:

- [Data persistence](#) [► 162]
- [Resetting the PLC project](#) [► 218]

16.2.13 SUPER

SUPER is a special variable used for object-oriented programming.

SUPER is the pointer of a function block to the basic function block instance from which the function block was created. The SUPER pointer thus also allows access to the implementation of the methods of the basic function block (basic class). A SUPER pointer is automatically available for each function block.

You can only use SUPER in method implementations and associated function block implementations.

Dereferencing the pointer: SUPER^

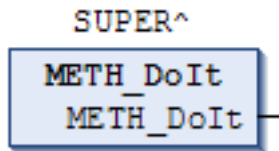
Using the SUPER pointer: Use the SUPER keyword to invoke the implementation of the base class or a method that is valid in the instance of the base class.

Examples:

ST:

```
SUPER^(); // Call of FB-body of base class
SUPER^.METH_DoIt(); // Call of method METH_DoIt that is implemented in base class
```

FBD/CFC/LD:



SUPER is not implemented for Instruction List (IL).

Using the SUPER and THIS pointers:

Function block FB_Base:

```
FUNCTION_BLOCK FB_Base
VAR_OUTPUT
    nCnt : INT;
END_VAR
```

Method FB_Base.METH_DoIt:

```
METHOD METH_DoIt : BOOL
nCnt := -1;
```

Method FB_Base.METH_DoAlso:

```
METHOD METH_DoAlso : BOOL
METH_DoAlso := TRUE;
```

Function block FB_1:

```
FUNCTION_BLOCK FB_1 EXTENDS FB_Base
VAR_OUTPUT
    nBase: INT;
END_VAR

THIS^.METH_DoIt(); // Call of the methods of FB_1
THIS^.METH_DoAlso();

SUPER^.METH_DoIt(); // Call of the methods of FB_Base
SUPER^.METH_DoAlso();
nBase := SUPER^.nCnt;
```

Method FB_1.METH_DoIt:

```
METHOD METH_DoIt : BOOL
nCnt := 1111;
METH_DoIt := TRUE;
```

Method FB_1.METH_DoAlso:

```
METHOD METH_DoAlso : BOOL
nCnt := 123;
METH_DoAlso := FALSE;
```

Program MAIN:

```
PROGRAM MAIN
VAR
    fbMyBase : FB_Base;
```

```

    fbMyFB_1 : FB_1;
    nTHIS    : INT;
    nBase    : INT;
END_VAR

fbMyBase();
nBase := fbmyBase.nCnt;
fbMyFB_1();
nTHIS := fbMyFB_1.nCnt;

```

See also:

- [Data types \[► 756\]](#)
- [THIS \[► 694\]](#)

16.2.14 THIS

THIS is a special variable used for object-oriented programming.

THIS is the pointer of a function block to its own function block instance. A THIS pointer is automatically available for each function block.

You can only use THIS in methods and function blocks. THIS is available for implementation in the **input assistant** in the **Keywords** category.

Dereferencing the pointer: THIS^

Using the THIS pointer:

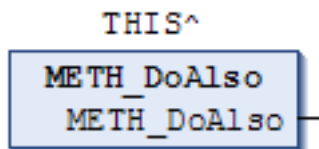
- If a function block variable is shadowed by a local variable within a method, the THIS pointer can be used to set the function block variable. See example (1) below
- The THIS pointer can also be used to assign a pointer to the own FB instance with a function call. (See example (2) below)

Examples:

ST:

```
THIS^.METH_DoIt();
```

FBD/CFC/LD:



THIS is not implemented for Instruction List (IL).

Examples:

(1) The function block variable nVarB is shadowed by the local variable nVarB.

```

FUNCTION_BLOCK FB_A
VAR_INPUT
    nVarA: INT;
END_VAR

nVarA := 1;

FUNCTION_BLOCK FB_B EXTENDS FB_A
VAR_INPUT
    nVarB : INT := 0;
END_VAR

nVarA := 11;

```

```
nVarB := 2;

METHOD DoIt : BOOL
VAR_INPUT
END_VAR
VAR
    nVarB : INT;
END_VAR

nVarB := 22; // The local variable nVarB is set.
THIS^.nVarB := 222; // The function block variable nVarB is set even though nVarB is obscured.

PROGRAM MAIN
VAR
    fbMyfbB : FB_B;
END_VAR

fbMyfbB(nVarA:=0, nVarB:= 0);
fbMyfbB.DoIt();
```

(2) A function call needs the reference to the own FB instance.

```
FUNCTION F_FunA
VAR_INPUT
    fbMyFbA : FB_A;
END_VAR
...;

FUNCTION_BLOCK FB_A
VAR_INPUT
    nVarA: INT;
END_VAR
...;

FUNCTION_BLOCK FB_B EXTENDS FB_A
VAR_INPUT
    nVarB: INT := 0;
END_VAR

nVarA := 11;
nVarB := 2;

METHOD DoIt : BOOL
VAR_INPUT
END_VAR
VAR
    nVarB: INT;
END_VAR

nVarB := 22; //The local variable nVarB is set.
F_FunA(fbMyFbA := THIS^); //F_FunA is called via THIS^.

PROGRAM MAIN
VAR
    fbMyFbB: FB_B;
END_VAR

fbMyFbB(nVarA:=0 , nVarB:= 0);
fbMyFbB.DoIt();
```

See also:

- [Pointer \[► 767\]](#)
- [SUPER \[► 692\]](#)

16.2.15 Variable types - attribute keywords

In the variable declaration, you can add the following keywords to the variable type:

- **RETAIN:** for remanent variables of type RETAIN
- **PERSISTENT:** for remanent variables of type PERSISTENT
- **CONSTANT:** for constants

See also:

- [Constants \[▶ 744\]](#)
- [Remanent variables - RETAIN, PERSISTENT \[▶ 691\]](#)

16.3 Operators

TwinCAT 3 PLC supports all operators of the IEC 61131-3 standard. These operators are implicitly known throughout the project. In addition to the IEC operators, TwinCAT 3 PLC supports some operators which are not described in the IEC 61131-3 standard.

Operators are used in a function block like functions.

● Operations with floating point data types

i For operations with floating point data types, the result depends on the target system hardware used.

● Operations with overflow or underflow

i For operations with overflow or underflow in the data type, the calculation result depends on the target system hardware used.

i Information about the processing order (binding strength) of the ST operators can be found in section "[ST Expressions \[▶ 625\]](#)".

Overflow/underflow in the data type

The TwinCAT 3 compiler generates code for the respective target device and always calculates intermediate results with the native size specified by the target system. For example, on x86 and ARM systems at least 32-bit intermediate values are used, while on x64 systems 64-bit intermediate values are always used. This offers significant advantages in computing speed and often produces the expected result. However, it also means that an overflow or underflow in the data type may not be truncated.

Sample 1

The result of this addition is not truncated and the result in dwVar is 65536.

```
VAR
  nVarWORD   : WORD;
  nVarDWORD  : DWORD;
END_VAR

nVarWORD := 65535;
nVarDWORD := nVarWORD + 1;
```

Sample 2

The overflow and underflow in the data type is not truncated, and the results (bVar1, bVar2) of both comparisons are FALSE on 32-bit and on 64-bit hardware.

```
VAR
  nVar1 : WORD;
  nVar2 : WORD;
  bVar1 : BOOL;
  bVar2 : BOOL;
END_VAR

nVar1 := 65535;
nVar2 := 0;
bVar1 := (nVar1 + 1) = nVar2;
bVar2 := (nVar2 - 1) = nVar1;
```

Sample 3

The assignment to nVar3 truncates the value to the target data type WORD, and the result bVar1 is TRUE.

```
VAR
  nVar1 : WORD;
  nVar2 : WORD;
```

```
nVar3 : WORD;  
bVar1 : BOOL;  
END_VAR  
  
nVar1 := 65535;  
nVar2 := 0;  
nVar3 := (nVar1 + 1);  
bVar1 := (nVar3 = nVar2);
```

Sample 4

A conversion can be added to force the compiler to truncate the intermediate result.

The type conversion ensures that both comparisons only compare 16 bits and that the results (bVar1, bVar2) of both comparisons are TRUE.

```
VAR  
nVar1 : WORD;  
nVar2 : WORD;  
bVar1 : BOOL;  
bVar2 : BOOL;  
END_VAR  
  
nVar1 := 65535;  
nVar2 := 0;  
bVar1 := (TO_WORD(nVar1 + 1) = nVar2);  
bVar2 := (TO_WORD(nVar2 - 1) = nVar1);
```

Address operators

- [ADR \[► 700\]](#)
- [BITADR \[► 701\]](#)
- [Content operator \[► 700\]](#)

Arithmetic operators

- [ADD \[► 701\]](#)
- [SUB \[► 702\]](#)
- [MUL \[► 703\]](#)
- [DIV \[► 704\]](#)
- [MOD \[► 705\]](#)
- [MOVE \[► 706\]](#)
- [INDEXOF \[► 706\]](#)
- [SIZEOF \[► 706\]](#)
- [XSIZEOF \[► 707\]](#)

Call operators

- [CAL \[► 707\]](#)

Selection operators

- [LIMIT \[► 708\]](#)
- [MAX \[► 708\]](#)
- [MIN \[► 708\]](#)
- [MUX \[► 709\]](#)
- [SEL \[► 709\]](#)

Bitshift Operators

- [ROL \[► 711\]](#)
- [ROR \[► 712\]](#)

- [SHL \[▶ 710\]](#)
- [SHR \[▶ 711\]](#)

Bitstring operators

- [AND \[▶ 712\]](#)
- [AND THEN \[▶ 713\]](#)
- [NOT \[▶ 713\]](#)
- [OR \[▶ 713\]](#)
- [OR ELSE \[▶ 714\]](#)
- [XOR \[▶ 714\]](#)

Namespace operators

The namespace operators are an extension of the IEC 61131-3 operators. They offer options for making the access to variables or modules unique, even if you use the same variable or module name several times in the project.

- [Library namespace \[▶ 715\]](#)
- [Enumeration namespace \[▶ 716\]](#)
- [Global namespace \[▶ 715\]](#)
- [Namespace for Global Variables Lists \[▶ 715\]](#)

Numeric operators

- [ABS \[▶ 716\]](#)
- [ACOS \[▶ 716\]](#)
- [ASIN \[▶ 717\]](#)
- [ATAN \[▶ 717\]](#)
- [COS \[▶ 717\]](#)
- [EXP \[▶ 718\]](#)
- [EXPT \[▶ 718\]](#)
- [LN \[▶ 719\]](#)
- [LOG \[▶ 719\]](#)
- [SIN \[▶ 719\]](#)
- [SQRT \[▶ 720\]](#)
- [TAN \[▶ 720\]](#)

Type conversion operators

You can explicitly call type conversion operators. For typed conversions from one elementary type to another elementary type and also for overloads, the type conversion operators described below are available. Conversions from a "smaller" type to a "larger" type, such as from BYTE to INT or from WORD to DINT, are also possible implicitly.

Typed conversion: `<elementary data type>_TO_<another elementary data type>`

Overloaded conversion: `TO_<elementary data type>`

Elementary data types:

```
<elementary data type> =
__UXINT | __XINT | __XWORD | BIT | BOOL | BYTE | DATE | DINT | DT | DWORD | INT | LDATE | LDT | LINT
| LREAL | LTIME | LTOD | LWORD | REAL | SINT | TIME | TOD | UDINT | UINT | ULINT | USINT | WORD
```

The keywords `TIME_OF_DAY` and `DATE_AND_TIME` are alternative notations for the data types `TOD` and `DT`. `TIME_OF_DAY` and `DATE_AND_TIME` are not mapped as a type conversion command.

i For a type conversion operator, if the operand value is outside the value range of the target data type, the result output is undefined. This is the case, for example, when a negative operand value is converted from `LREAL` to the target data type `UINT`.

Information may be lost during type conversion from larger to smaller types.

i **String manipulation when converting to `STRING` or `WSTRING`**

With a type conversion to `STRING` or `WSTRING`, the typed value is stored as a left aligned string and truncated if it is overlong. Therefore, declare the return variables for the type conversion operators `<type>_TO_STRING` and `<type>_TO_WSTRING` long enough to accommodate the string without manipulation.

- [Overloading \[▶ 721\]](#)
- [<INT type> TO <INT type> \[▶ 724\]](#)
- [<type> TO `BOOL` \[▶ 722\]](#)
- [`BOOL` TO <type> \[▶ 723\]](#)
- [`DATE/DT` TO <type> \[▶ 728\]](#)
- [`REAL/LREAL` TO <type> \[▶ 725\]](#)
- [`STRING` TO <type> \[▶ 725\]](#)
- [`TO_STRING/TO_WSTRING` for enumeration variables \[▶ 726\]](#)
- [`TIME/TOD` TO <type> \[▶ 727\]](#)
- [`TRUNC` \[▶ 729\]](#)
- [`TRUNC INT` \[▶ 729\]](#)

Comparison operators

The comparison operators are Boolean operators, which compare two inputs (first and second operand).

- [EQ \[▶ 730\]](#)
- [GE \[▶ 730\]](#)
- [GT \[▶ 730\]](#)
- [LE \[▶ 731\]](#)
- [LT \[▶ 731\]](#)
- [NE \[▶ 731\]](#)

Further operators

- [_DELETE \[▶ 734\]](#)
- [_ISVALIDREF \[▶ 735\]](#)
- [_NEW \[▶ 732\]](#)
- [_QUERYINTERFACE \[▶ 735\]](#)
- [_QUERYPOINTER \[▶ 737\]](#)
- [_VARINFO \[▶ 741\]](#)
- [_POUNAME \[▶ 743\]](#)
- [_POSITION \[▶ 743\]](#)

16.3.1 Address operators

16.3.1.1 ADR

The operator is an extension of the IEC 61131-3 standard.

ADR delivers the address of its argument in a PVOID or, depending on the runtime system, in a DWORD (32-bit systems) or LWORD (32 and 64-bit systems). To ensure independence from the runtime system architecture, it is recommended to use PVOID as data type.

⚠ CAUTION

Shifting the contents of addresses by Online Change

If you use an online change, contents of addresses may shift. As a result, POINTER variables could point to an invalid memory area. In order to avoid problems, make sure that TwinCAT updates the value of pointers when an online change takes place.

Syntax:

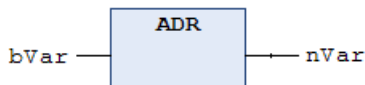
```
VAR
  <address name> : PVOID | DWORD | LWORD | __XWORD | POINTER TO < basis data type>;
END_VAR
<address name> := ADR(<variable name>);
```

Samples:

ST:

```
nVar := ADR(bVAR);
```

FBD:



Sample with different data types:

```
FUNCTION_BLOCK FB_Address
VAR
  nVar      : INT := 10;
  pNumber   : POINTER TO INT;
  nAddress1 : PVOID;
  nAddress2 : DWORD;
  nAddress3 : LWORD;
  nAddress4 : __XWORD;
END_VAR

pNumber := ADR(nVar); // pNumber wird der Adresse von nVar zugewiesen
nAddress1 := ADR(nVar); // PVOID : für 32- und 64-Bit-Systeme
nAddress2 := ADR(nVar); // DWORD  : nur für 32-Bit-Systeme
nAddress3 := ADR(nVar); // LWORD  : für 32- und 64-Bit-Systeme
nAddress4 := ADR(nVar); // __XWORD : für 32- und 64-Bit-Systeme
```

See also:

- [POINTER \[► 767\]](#)
- [Special data types XINT, UXINT, XWORD and PVOID \[► 767\]](#)

16.3.1.2 Content operator

The operator is an extension of the IEC 61131-3 standard.

The operator allows dereferencing of a pointer. Append the operator as ^ on the pointer identifier.

⚠ CAUTION

Shift of contents of addresses through online change

If you use an online change, contents of addresses may shift.

Example:

ST:

```
pSample : POINTER TO INT;
nInt1   : INT;
nInt2   : INT;
pSample := ADR(nInt1);
nInt2   := pSample^;
```

16.3.1.3 BITADR

The operator is an extension of the IEC 61131-3 standard.

BITADR returns the bit offset within the segment in a DWORD.

The highest-value nibble (4 bits) in this DWORD describes the memory area:

Flags: 16x40000000

Input: 16x80000000

Output: 16xC0000000

⚠ CAUTION

Shift of contents of addresses through online change
 If you use an online change, contents of addresses may shift.

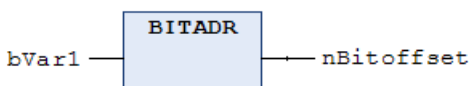
Examples:

ST:

```
VAR
    bVar1 AT %IX2.3 : BOOL;
    nBitoffset : DWORD;
END_VAR

nBitoffset := BITADR(bVar1); (*Result if byte addressing=TRUE: 16x80000013, if byte addressing = FALSE : 16x80000023*)
```

FBD:



16.3.2 Arithmetic operators

16.3.2.1 ADD

The IEC operator adds variables.

Allowed data types: __UXINT, __XINT, __XWORD, BYTE, DATE, DATE_AND_TIME, DINT, DT, DWORD, INT, LDATE, LDATE_AND_TIME, LDT, LINT, LREAL, LTIME, LTOD, LWORD, REAL, SINT, TIME, TIME_OF_DAY, TOD, UDINT, UINT, ULINT, USINT, WORD

Possible combinations for time data types:

- TIME + TIME = TIME
- TIME + LTIME = LTIME
- LTIME + LTIME = LTIME

Possible combinations for date and time data types:

- TOD + TIME = TOD
- DT + TIME = DT

- TOD + LTIME = LTOD
- DT + LTIME = LDT
- LTOD + TIME = LTOD
- LDT + LTIME = LDT
- LTOD + LTIME = LTOD
- LDT + LTIME = LDT

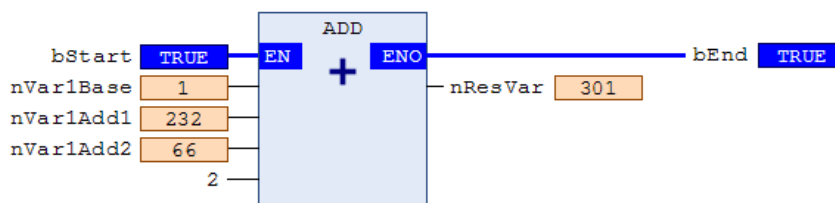
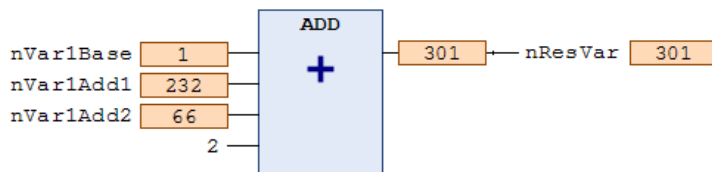
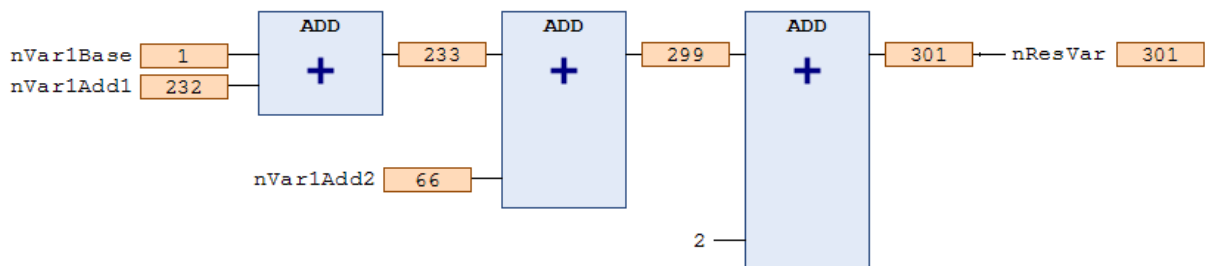
Special feature in the FBD/LD editor: You can extend the ADD operator with function block inputs. The number of additional function block inputs is limited.

Samples:

ST:

```
nVar := 7+2+4+7;
```

FBD:



16.3.2.2 SUB

The IEC operator subtracts variables.

Allowed data types: BYTE, WORD, DWORD, LWORD, SINT, USINT, INT, UINT, DINT, UDINT, LINT, ULINT, REAL, LREAL, TIME, TIME_OF_DAY, TOD, LTIME, LTIME_OF_DAY, LTOD, DATE, DATE_AND_TIME, DT, LDATE, LDATE_AND_TIME, LDT

Possible combinations for time data types:

- TIME - TIME = TIME
- LTIME - LTIME = LTIME

Possible combinations for date and time data types:

- DATE - DATE = TIME
- LDATE - LDATE = LTIME

- TOD - TIME = TOD
- LTOD - LTIME = LTOD
- TOD - TOD = TIME
- LTOD - LTOD = LTIME
- DT - TIME = DT
- LDT - LTIME = LDT
- DT - DT = TIME
- LDT - LDT = LTIME



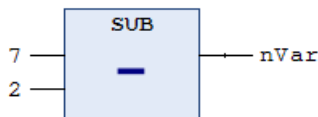
Negative TIME/LTIME values are undefined.

Samples:

ST:

```
nVar := 7-2;
```

FBD:



16.3.2.3 MUL

The IEC operator is used for multiplication of variables.

Permitted data types: BYTE, WORD, DWORD, LWORD, SINT, USINT, INT, UINT, DINT, UDINT, LINT, ULINT, REAL, LREAL, TIME

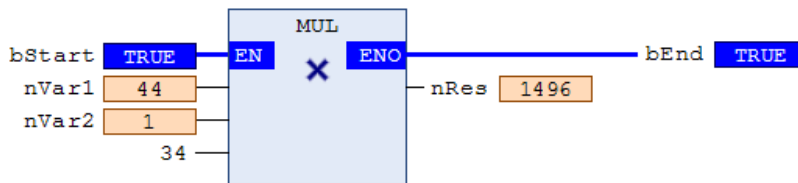
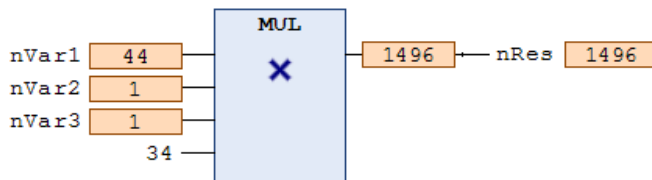
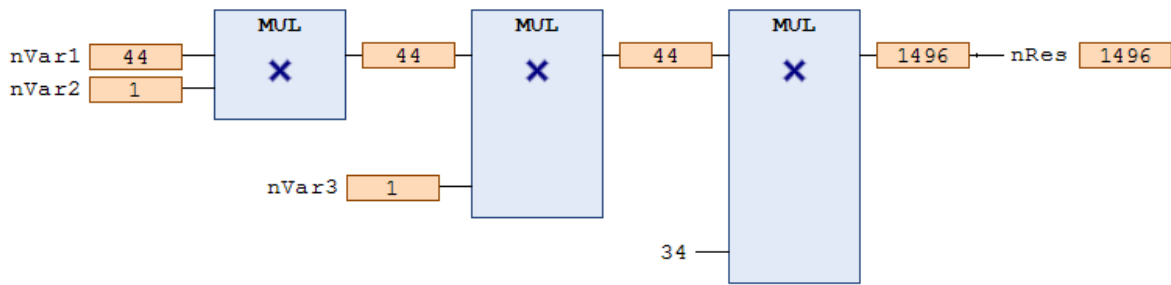
Special feature in the FBD/LD Editor: You can extend the MUL operator with additional function block inputs. The number of additional function block inputs is limited.

Examples:

ST:

```
nVar := 7*2*4*7;
```

FBD:



16.3.2.4 DIV

The IEC operator is used for division of variables.

Permitted data types: BYTE, WORD, DWORD, LWORD, SINT, USINT, INT, UINT, DINT, UDINT, LINT, ULINT, REAL, LREAL, TIME



In TwinCAT, division by zero always leads to an exception and the corresponding task is stopped.

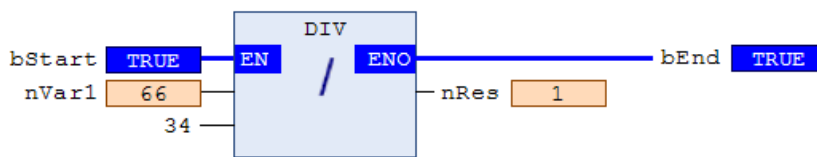
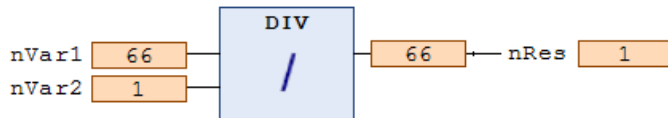
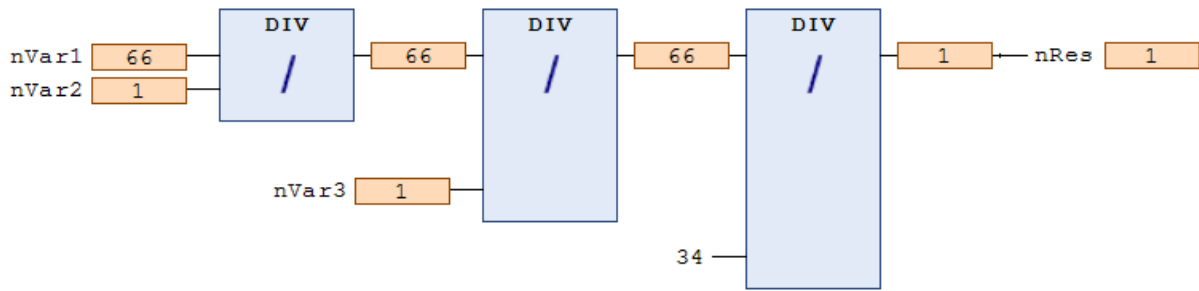
Examples:

ST:

```
nVar := 8/2;
```

FBD:

1. Series of DIV function blocks, 2. Individual DIV function block, 3. DIV function block with EN/ENO parameters



Note the possibility to monitor for division by 0 during runtime using the implicit monitoring functions CheckDivInt, CheckDivLint, CheckDivLReal and CheckDivLReal.

See also:

- [Division checks \(POUs CheckDivDInt, CheckDivLint, CheckDivReal, CheckDivLReal\)](#) [► 166]
- POU CheckDivLint
- POU CheckDivLReal
- POU CheckDivReal

16.3.2.5 MOD

The IEC operator is used for modulo division.

The result of the function is the integer remainder of the division.

Permitted data types: BYTE, WORD, DWORD, LWORD, SINT, USINT, INT, UINT, DINT, UDINT, LINT, ULINT



In TwinCAT, division by zero always leads to an exception and the corresponding task is stopped. However, the result of Mod 0 is 0.

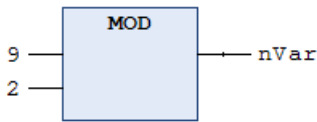
Samples:

Result: nVar is 1.

ST:

```
nVar := 9 MOD 2;
```

FBD:



16.3.2.6 MOVE

IEC-operator is used to assign a variable to another variable of a corresponding type.

Since MOVE is available as a function block in the CFC, FBD and LD editors, you can apply the EN/ENO functionality to a variable assignment there.

Examples:

Result: var2 receives the value of var1.

CFC in conjunction with the EN/ENO function:

If en_i is TRUE, TwinCAT assigns the value of variable var1 to variable var2.



ST:

```
ivar2 := MOVE(ivar1);
```

This corresponds to:

```
ivar2 := ivar1;
```

16.3.2.7 INDEXOF

The operator is an extension of the IEC 61131-3 standard.

Instead of the INDEXOF operator, the ADR operator is available in TwinCAT to obtain a pointer to the index of a function block.

See also:

- [ADR \[▶ 700\]](#)

16.3.2.8 SIZEOF

The operator is an extension of the IEC 61131-3 standard.

The operator is used to determine the number of bytes required by the specified variable x. The SIZEOF operator always returns an unsigned value. The type of the return variable adapts to the detected size of variable x.

Return value of SIZEOF(x)	Data type of the constant, which TwinCAT implicitly uses for the size that was found.
0 <= size of x < 256	USINT
256 <= size of x < 65536	UINT
65536 <= size of x < 4294967296	UDINT
4294967296 <= size of x	ULINT

Samples:

Result: nVar is 10.

ST:

```
aArr1 : ARRAY[0..4] OF INT;
nVar  : INT;

nVar := SIZEOF(aArr1); (*nVar := USINT#10;*)
```

Also see about this

 [XSIZEOF \[▶ 707\]](#)

16.3.2.9 XSIZEOF



Available from TC3.1 Build 4026

The operator is an extension of the IEC 61131-3 standard.

The XSIZEOF operator determines the number of bytes needed in the passed variable or data type.

An unsigned value is always returned. The data type of the return value <return value> is specified as follows: on 64-bit platforms the type is ULINT, on all other platforms it is UDINT. To generate code that runs on all platforms, the return value can be declared with the `__UXINT` data type.

Syntax:

```
<return value> := XSIZEOF( <variable> );
```

Sample:

```
PROGRAM MAIN
VAR
  nReturnValue : __UXINT;           // Datentyp bei 64-bit-Plattformen: ULINT
  aData1      : ARRAY[0..4] OF INT;
END_VAR

nReturnValue := XSIZEOF(aData1);
```

Result:

```
nReturnValue = 10
```



When assigning to a variable of type `__UXINT` it is advisable to use the operator XSIZEOF instead of the operator [SIZEOF \[▶ 706\]](#). This is because with XSIZEOF the data type of the return value depends on the platform. As a result, problems that occur when using the SIZEOF operator do not occur.

16.3.3 Call operators

16.3.3.1 CAL

The IEC operator is used to call a function block.

CAL calls the instance of a function block in IL.

Syntax:

```
CAL <function block> (<input variable 1> := <value>, <input variable N> := <value>)
```

Example:

Call of the instance fbInst of a function block with assignment of the input variables nVar1, bVar2 to 0 or TRUE.

```
CAL fbInst(nVar1 := 0, bVar2 := TRUE)
```

16.3.4 Selection operators

16.3.4.1 LIMIT

The IEC selection operator is used for limitation.

Syntax: OUT := LIMIT(Min, IN, Max)

means: OUT := MIN(MAX (IN, Min), Max)

Max is the upper bound, Min is the lower bound for the result. If the value IN exceeds the upper bound Max, LIMIT returns Max. If IN falls below Min, the result is Min.

Permitted data types for IN and OUT: all

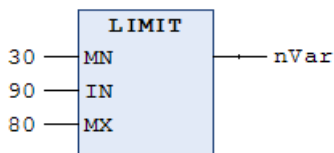
Examples:

Result: nVar is 80.

ST:

```
nVar := LIMIT(30, 90, 80);
```

FBD:



16.3.4.2 MAX

The IEC operator is used for the maximum function. It returns the largest of the transferred input values.

Syntax: OUT := MAX(IN0, IN1, <further inputs>)

Permitted data types: all

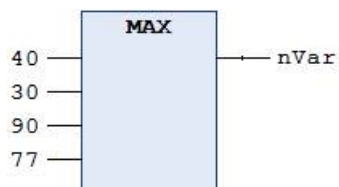
Samples:

Result: nVar is 90.

ST:

```
nVar := MAX(40, 30, 90, 77);
```

FBD:



16.3.4.3 MIN

The IEC operator is used for the minimum function. It returns the smallest of the transferred input values.

Syntax: OUT := MIN(IN0, IN1, <further inputs>)

Permitted data types: all

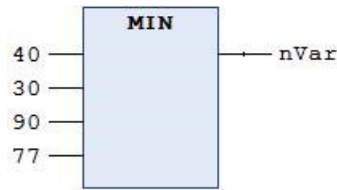
Samples:

Result: nVar is 30.

ST:

```
nVar := MIN(40, 30, 90, 77);
```

FBD:



16.3.4.4 MUX

The IEC operator is used as a multiplexer.

Syntax: OUT := MUX(K, IN0,...,INn)

This means OUT = IN_K

Permitted data types for K: BYTE, WORD, DWORD, LWORD, SINT, USINT, INT, UINT, DINT, LINT, ULINT, UDINT

IN0, ..., INn and OUT: Any identical data type. Make sure that variables with the same type are used at all three positions, especially when using user-defined data types. The compiler checks the type equality and issues compilation errors. In particular, the allocation of instances of a function block to interface (variables) is not supported.

MUX selects the K-th value from a set of values. The first value corresponds to K=0. If K is greater than the number of additional inputs (n), TwinCAT passes on the last value (INn).



For the purpose of run-time optimization, TwinCAT only calculates the expression with which you have preceded IN_K.

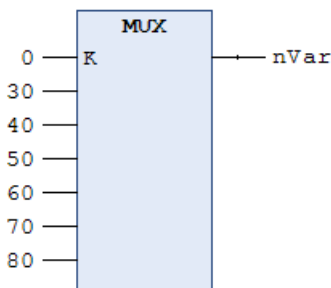
Examples:

Result: nVar is 30.

ST:

```
nVar := MUX(0, 30, 40, 50, 60, 70, 80);
```

FBD:



16.3.4.5 SEL

The IEC operator is used for bitwise selection.

Syntax:

OUT := SEL(G, IN0, IN1) means:

OUT := IN0; if G = FALSE

OUT := IN1; if G = TRUE

Permitted data types:

IN0, ..., INn and OUT: Any identical data type. Make sure that variables with the same type are used at all three positions, especially when using user-defined data types. The compiler checks the type equality and issues compilation errors. In particular, the assignment of instances of a function block to interface (variables) is not supported.

G: BOOL



TwinCAT does not compute an expression that precedes IN0 if G = TRUE. TwinCAT does not compute an expression that precedes IN1 if G = FALSE.

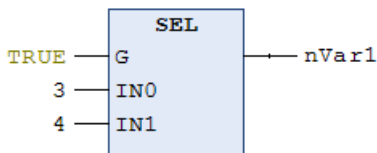
With graphical programming languages, the expressions at IN0 and IN1 are calculated independently of the input G if a function block, a jump, a return, a branch or an edge detection is connected upstream.

Samples:

ST:

```
nVar1 := SEL(TRUE,3,4); (*Result: 4*)
```

FBD:



16.3.5 Bitshift operators

16.3.5.1 SHL

The IEC operator is used for bitwise shifting of an operand to the left.

Syntax: erg := SHL (in, n)

in: Operand that is moved to the left.

n: Number of bits in move to the left.



If n exceeds the width of the data type, it depends on the target system how BYTE, WORD, DWORD and LWORD operands are populated. The target systems cause padding with zeros or with n MOD <register width>.



Note that the number of bits that TwinCAT takes into account for the arithmetic operation is determined by the data type of the input variable.

Examples:

The results for nResByte and nResWord are different, despite the fact that the values of the input variables nInByte and nInWord are the same, because the data types of the input variables are different.

ST:

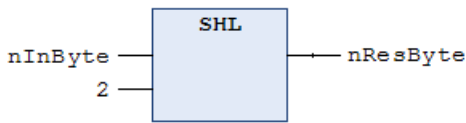
```
PROGRAM Shl_st
VAR
  nInByte   : BYTE:=16#45; (*2#01000101*)
  nInWord   : WORD:=16#0045; (*2#0000000001000101*)
  nResByte  : BYTE;
  nResWord  : WORD;
  nVar      : BYTE := 2;
```

```

END_VAR
nResByte := SHL(nInByte,nVar); (*Result is 16#14, 2#00010100*)
nResWord := SHL(nInWord,nVar); (*Result is 16#0114, 2#0000000100010100*)

```

FBD:



16.3.5.2 SHR

The IEC operator is used for bitwise shifting of an operand to the right.

Syntax: erg := SHR (in, n)

in: Operand that is moved to the right.

n: Number of bits in move to the right.

i If n exceeds the width of the data type, it depends on the target system how BYTE, WORD, DWORD and LWORD operands are populated. The target systems cause padding with zeros or with n MOD <register width>.

Examples:

ST:

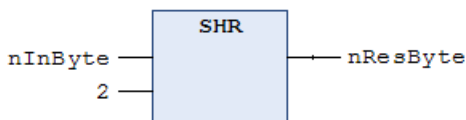
```

PROGRAM Shr_st
VAR
  nInByte   : BYTE:=16#45; (*2#01000101*)
  nInWord   : WORD:=16#0045; (*2#0000000001000101*)
  nResByte  : BYTE;
  nResWord  : WORD;
  nVar      : BYTE := 2;
END_VAR

nResByte := SHR(nInByte,nVar); (*Result is 16#11, 2#00010001*)
nResWord := SHR(nInWord,nVar); (*Result is 16#0011, 2#0000000000010001*)

```

FBD:



16.3.5.3 ROL

The IEC operator is used for bitwise rotation of an operand to the left.

Permitted data types: BYTE, WORD, DWORD, LWORD

Syntax: erg := ROL (in, n)

TwinCAT moves n times to the left by 1 bit and at the same time adds the bit with the extreme left position on the right.

i The number of bits, which TwinCAT takes into account for the calculation, is determined by the data type of the input variable. If this is a constant, TwinCAT considers the smallest possible data type. The data type of the output variable has no effect on the arithmetic operation.

Examples:

The results for nResByte and nResWord are different, depending on the data type of the input variable, despite the fact that the values of the input variables nInByte and nInWord are the same.

ST:

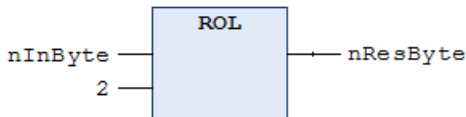
```

PROGRAM Rol_st
VAR
  nInByte  : BYTE := 16#45;
  nInWord  : WORD := 16#45;
  nResByte : BYTE;
  nResWord : WORD;
  nVar     : BYTE := 2;
END_VAR

nResByte := ROL(nInByte,nVar); (*Result: 16#15*)
nResWord := ROL(nInWord,nVar); (*Result: 16#0114*)

```

FBD:



16.3.5.4 ROR

The IEC operator is used for bitwise rotation of an operand to the right.

Permitted data types: BYTE, WORD, DWORD, LWORD

Syntax: erg := ROR (in, n)

TwinCAT moves n times to the right by 1 bit and at the same time adds the bit with the extreme right position on the left.



The number of bits, which TwinCAT takes into account for the calculation, is determined by the data type of the input variable. If this is a constant, TwinCAT considers the smallest possible data type. The data type of the output variable has no effect on the arithmetic operation.

Examples:

The results for nResByte and nResWord are different, depending on the data type of the input variable, despite the fact that the values of the input variables nInByte and nInWord are the same.

ST:

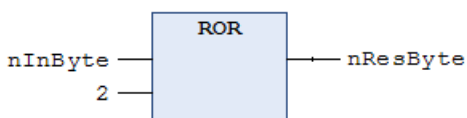
```

PROGRAM Ror_st
VAR
  nInByte  : BYTE:=16#45;
  nInWord  : WORD:=16#45;
  nResByte : BYTE;
  nResWord : WORD;
  nVar     : BYTE :=2;
END_VAR

nResByte := ROR(nInByte,nVar); (*Result: 16#51*)
nResWord := ROR(nInWord,nVar); (*Result: 16#4011*)

```

FBD:



16.3.6 Bitstring operators

16.3.6.1 AND

The IEC operator is used for bitwise AND of bit operands.

If the input bits are 1, the output bit is 1, otherwise 0.

Permitted data types: BOOL, BYTE, WORD, DWORD, LWORD

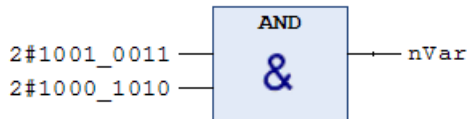
Examples:

Result: nVar is 2#1000_0010.

ST:

```
nVar := 2#1001_0011 AND 2#1000_1010
```

FBD:



16.3.6.2 AND_THEN

The operator is an extension of the IEC 61131-3 standard.

The operator AND_THEN is only allowed for programming in Structured Text with the following operation: AND operation of operands of type BOOL and BIT, with short-circuit evaluation. This means:

If all operands are TRUE, the result of the operation is also TRUE, otherwise it is FALSE.

But: TwinCAT also executes the expressions for other operands only if the first operand of the AND_THEN operator is TRUE. In conditions such as IF (ptr < > 0 AND_THEN ptr ^ = 99) THEN..., this can avoid problems with null pointers.

In contrast, TwinCAT always evaluates all operands if the IEC operator AND is used.

See also:

- [AND \[▶ 712\]](#)

16.3.6.3 NOT

The IEC operator is used for bitwise NOT of a bit operand.

If the corresponding input bit is 0, the output bit is 1 and vice versa.

Permitted data types: BOOL, BYTE, WORD, DWORD, LWORD

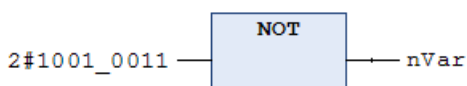
Examples:

Result: nVar is 2#0110_1100.

ST:

```
nVar := NOT 2#1001_0011
```

FBD:



16.3.6.4 OR

The IEC operator is used for bitwise OR of bit operands.

If at least one of the input bits is 1, the output bit is 1, otherwise 0.

Permitted data types: BOOL, BYTE, WORD, DWORD, LWORD

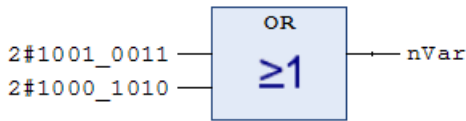
Examples:

Result: nVar is 2#1001_1011.

ST:

```
nVar := 2#1001_0011 OR 2#1000_1010
```

FBD:



16.3.6.5 OR_ELSE

The operator is an extension of the IEC 61131-3 standard.

The operator OR_ELSE is only allowed for programming in Structured Text: OR operation of operands of type BOOL and BIT, with short-circuit evaluation.

This means:

If at least one of the operands is TRUE, the result of the operation is also TRUE, otherwise it is FALSE.

In contrast to using the IEC operator OR, with OR_ELSE the expressions for all further operands are no longer executed, if one of the operands was evaluated with TRUE.

Example:

```
Function_Block FB_Sample
VAR
    nCounter : INT;
END_VAR

METHOD TestMethod : BOOL
nCounter := nCounter + 1;
TestMethod := TRUE;

PROGRAM MAIN
VAR
    fbSampleOr : FB_Sample;
    fbSampleOrElse : FB_Sample;
    bResult : BOOL;
    bVar : BOOL;
END_VAR

bResult : bVar OR fbSampleOr.TestMethod();
// Counter of fbSampleOr increases as the method is executed

bResult := bVar OR_ELSE fbSampleOrElse.TestMethod();
//Counter of fbSampleOrElse does not increases as the method is not executed
```

bVar is TRUE, therefore the result bResult is also TRUE when using both OR and OR_ELSE.

The difference between the two logical operators is that the second operand (the calling of the method) is only executed when using OR. In this case the counter of the function block instance fbSampleOr is incremented.

Since the first operand bVar1 is already yielding TRUE, the second operand is no longer executed when using OR_ELSE, therefore the method of the function block instance fbSampleOrElse is not called and the counter is not incremented.

See also:

- [OR \[► 713\]](#)

16.3.6.6 XOR

The IEC operator is used for bitwise XOR operation of bit operands.

If only one of the two input bits returns the value 1, the output bit is set to 1. If both inputs are 1 or both are 0, the output becomes 0.

Permitted data types: BOOL, BYTE, WORD, DWORD, LWORD



Note the following behavior of XOR function block in extended form, i.e. if there are more than 2 inputs: TwinCAT compares the inputs in pairs and then compares the respective results (this corresponds to the standard, but not necessarily the expectation).

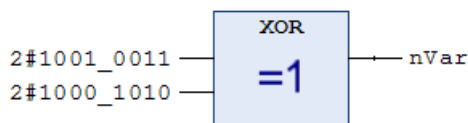
Examples:

Result: nVar is 2#0001_1001.

ST:

```
nVar := 2#1001_0011 XOR 2#1000_1010
```

FBD:



16.3.7 Namespace operators

The namespace operators are an extension of the IEC 61131-3 operators. They offer options for making the access to variables or modules unique, even if you use the same variable or module name several times in the project.

16.3.7.1 Global namespace

The operator is an extension of the IEC 61131-3 standard.

An instance path that starts with a dot . always opens a global namespace. So, if there is a local variable with the same name <varname> as a global variable, <varname> is used to address the global variable.

16.3.7.2 Namespace for Global Variables Lists

The operator is an extension of the IEC 61131-3 standard.

You can use the name of a global variable list (GVL) as namespace identifier for the variables defined in the list. This makes it possible to use variables with the same names in different global variable lists and still uniquely access a particular variable. The variable name should be preceded by the name of the global variable list, separated by a dot.

Syntax: <global variable list name>.<variable>

Example:

```
GVL1.nVar := GVL2.nVar;
```

The global variable lists GVL1 and GVL2 contain a variable nVar. TwinCAT copies the global variable nVar from the list GVL2 to nVar from the list GVL1.

An error message appears if you reference a variable that is declared in more than one global variable list without the preceding list name.

16.3.7.3 Library namespace

The operator is an extension of the IEC 61131-3 standard.

Syntax:

<library namespace>.<library function block identifier>

A library function block identifier is extended with the library namespace (prefixed with a dot as separator) to access the library function block in an unambiguous and qualified manner. Usually the namespace and the name of a library are the same.

Example:

A library is integrated in a project and contains a function block FB_Sample. However, a function block with the same name is already instantiated locally in the project. Use libA.fbSample as the name for the library block, in order to access the library block, rather than the local function block.

```
nVar1 := fbSample(nIn := 12); // Call of the project function block FB_Sample
nVar2 := lib.fbSample(nIn := 22); // Call of the library function block FB_Sample
```

You can define a different identifier for the namespace. To this end, in your capacity as library developer enter a namespace in the project information when you create a library project. Alternatively, you can specify a special namespace when you create a PLC project in the library manager for a library in the **Properties** view.

16.3.7.4 Enumeration namespace

The operator is an extension of the IEC 61131-3 standard.

You can use the TYPE name of an enumeration for unambiguous access to an enumeration constant. This enables you to use constants with same name in several enumerations.

The enumeration name is prefixed to the constant name, separated by a dot.

Syntax: < enumeration name>.<constant name>

Example:

The constant eBlue is a component of both the E_Colors enumeration and the E_Feelings enumeration.

```
eColor := E_Colors.eBlue; // Access to component Blue in enumeration Colors
eFeeling := E_Feelings.eBlue; // Acces to component Blue in enumeration Feeling
```

16.3.8 Numeric operators

16.3.8.1 ABS

The IEC operator returns the absolute value of a number.

Permitted data types for the input and output variables and number constants: any numeric basic data type

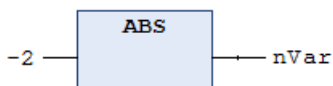
Examples:

Result: nVar is 2.

ST:

```
nVar := ABS(-2);
```

FBD:



16.3.8.2 ACOS

The IEC operator returns the arc cosine (inverse of cosine) for a number. The value is calculated in radians.

Permitted data types for the input variable, which specifies the angle in radians: any numeric basic data type

Permitted data types for the output variable: REAL and LREAL

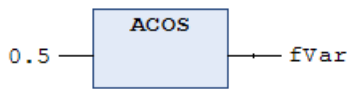
Examples:

Result: fVar is 1.0472.

ST:

```
fVar := ACOS(0.5);
```

FBD:



16.3.8.3 ASIN

The IEC operator returns the arc sine (inverse of sine) for a number.

Permitted data types for the input variable: Any numeric basic data type

Permitted data types for the output variables: REAL and LREAL

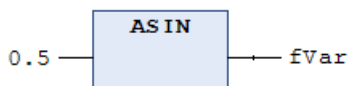
Examples:

Result: fVar is 0.523599.

ST:

```
fVar := ASIN(0.5);
```

FBD:



16.3.8.4 ATAN

The IEC operator returns the arc tangent (inverse of tangent) for a number. The value is calculated in radians.

Permitted data types for the input variable, which specifies the angle in radians: any numeric basic data type

Permitted data types for the output variables: REAL and LREAL

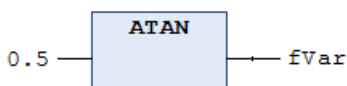
Examples:

Result: fVar is 0.463648.

ST:

```
fVar := ATAN(0.5);
```

FBD:



16.3.8.5 COS

The IEC operator returns the cosine value for a number.

Permitted data types for the input variable, which specifies the angle in radians: any numeric basic data type

Permitted data types for output variables: REAL and LREAL



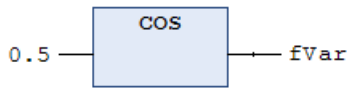
The allowed range for the input value is from -2^{63} to $+2^{63}$.
On x86 and x64 systems, if the input value is out of range, the function returns the input value.

Samples:

ST:

```
fVar := COS(0.5);
```

FBD:



Result: fVar is 0.877583.

16.3.8.6 EXP

The IEC operator returns the exponential function.

Permitted data types for the input variables: any numeric basic data type

Permitted data types for the output variables: REAL and LREAL

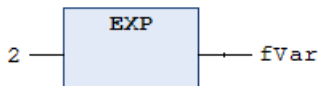
Examples:

Result: fVar is 7.389056099.

ST:

```
fVar := EXP(2);
```

FBD:

**16.3.8.7 EXPT**

The IEC operator exponentiates one number with another and returns the base to the power of the exponent: $\text{power} = \text{base}^{\text{exponent}}$. Both base and exponent are input values (parameters). The power function is not defined if the base is 0 and the exponent is negative at the same time. However, the behavior in this case depends on the platform.

Syntax: EXPT(<base>,<exponent>)

Permitted data types for the input values: Basic numerical data types (SINT, USINT, INT, UINT, DINT, UDINT, LINT, ULINT, REAL, LREAL, BYTE, WORD, DWORD, LWORD)

Permitted data types for the return value: Floating-point number types (REAL, LREAL).

If the result is stored in the REAL variable, a warning is generated for the conversion from LREAL to REAL.
--> return type is always LREAL.

For example, a warning is generated for the following PLC code:

```
real1 := EXPT(real2,INT#2);
```

Samples:

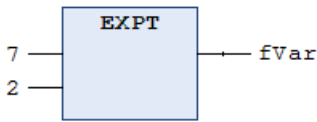
a) power function with literals

Result: fVar is 49.

ST:

```
fVar := EXPT(7,2);
```

FBD:



b) power function with variables

Result: fPow is 128.

```
PROGRAM MAIN
VAR
    fPow      : LREAL;
    nBase     : INT := 2;
    nExponent : INT := 7;
END_VAR
fPow := EXPT(nBase, nExponent);
```

16.3.8.8 LN

The IEC operator returns the natural logarithm of a number.

Permitted data types for the input variables: any numeric basic data type

Permitted data types for the output variables: REAL and LREAL

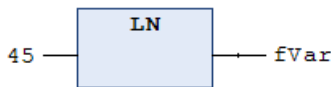
Examples:

Result: fVar is 3.80666.

ST:

```
fVar := LN(45);
```

FBD:



16.3.8.9 LOG

The IEC operator returns the base 10 logarithm of a number.

Permitted data types for the input variables: any numeric basic data type

Permitted data types for the output variable: REAL or LREAL

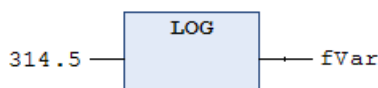
Examples:

Result: fVar is 2.49762.

ST:

```
fVar := LOG(314.5);
```

FBD:



16.3.8.10 SIN

The IEC operator returns the sine value for a number.

Permitted data types for the input variable, which specifies the angle in radians: any numeric basic data type

Permitted data types for output variables: REAL and LREAL



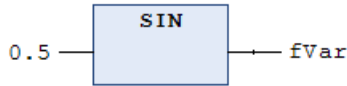
The allowed range for the input value is from -2^{63} to $+2^{63}$.
On x86 and x64 systems, if the input value is out of range, the function returns the input value.

Samples:

ST:

```
fVar := SIN(0.5);
```

FBD:



Result: fVar is 0.479426.

16.3.8.11 SQRT

The IEC operator returns the square root of a number.

Permitted data types for the input variables: any numeric basic data type

Permitted data types for the output variables: REAL or LREAL

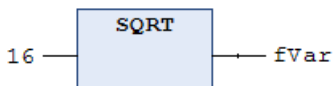
Examples:

Result: fVar is 4.

ST:

```
fVar := SQRT(16);
```

FBD:

**16.3.8.12 TAN**

The IEC operator returns the tangent value for a number.

Permitted data types for the input variable, which specifies the angle in radians: any numeric basic data type

Permitted data types for the output variables: REAL and LREAL

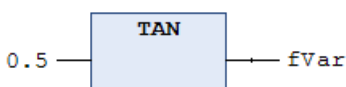
Examples:

Result: fVar is 0.546302.

ST:

```
fVar := TAN(0.5);
```

FBD:



16.3.9 Type conversion operators

You can explicitly call type conversion operators. For typed conversions from one elementary type to another elementary type and also for overloads, the type conversion operators described below are available. Conversions from a "smaller" type to a "larger" type, such as from BYTE to INT or from WORD to DINT, are also possible implicitly.

Typed conversion: `<elementary data type>_TO_<another elementary data type>`

Overloaded conversion: `TO_<elementary data type>`

Elementary data types:

```
<elementary data type> =
__UXINT | __XINT | __XWORD | BIT | BOOL | BYTE | DATE | DINT | DT | DWORD | INT | LDATE | LDT | LINT
| LREAL | LTIME | LTOD | LWORD | REAL | SINT | TIME | TOD | UDINT | UINT | ULINT | USINT | WORD
```

The keywords `TIME_OF_DAY` and `DATE_AND_TIME` are alternative notations for the data types `TOD` and `DT`. `TIME_OF_DAY` and `DATE_AND_TIME` are not mapped as a type conversion command.

i For a type conversion operator, if the operand value is outside the value range of the target data type, the result output is undefined. This is the case, for example, when a negative operand value is converted from `LREAL` to the target data type `UINT`.

Information may be lost during type conversion from larger to smaller types.

i **String manipulation when converting to `STRING` or `WSTRING`**
With a type conversion to `STRING` or `WSTRING`, the typed value is stored as a left aligned string and truncated if it is overlong. Therefore, declare the return variables for the type conversion operators `<type>_TO_STRING` and `<type>_TO_WSTRING` long enough to accommodate the string without manipulation.

See also:

- [Data types \[► 756\]](#)

16.3.9.1 Overloading

The operators convert values to other data types, explicitly specifying only a target data type and no initial data type (data type of the operand) ("overloaded conversion").

Overcharges are not part of IEC 61131-3. If you want to program strictly according to IEC61131-3, please use the operators of the `<type>_TO_<another type>` scheme described in the following sections.

Syntax:

```
<variable name> := <TO operator> ( <operand> );
<operand> = <variable name> | <literal>
```

Operators:

```
TO__UXINT
TO__XINT
TO__XWORD
TO_BIT
TO_BYTE
TO_BOOL
TO_DATE
TO_DINT
TO_DT
TO_DWORD
TO_INT
TO_LDATE
TO_LDT
TO_LINT
TO_LREAL
TO_LTIME
TO_LTOD
TO_LWORD
```

```

TO_REAL
TO_SINT
TO_STRING
TO_TIME
TO_TOD
TO_UDINT
TO_UINT
TO_ULINT
TO_USINT
TO_WORD
TO_WSTRING

```

The rules for typed conversions also apply to overloading.



If the input value of a type conversion operator is outside the value range of the output data type, the result of the operation is not defined and depends on the platform. This is the case, for example, if the input value for the conversion from LREAL to UDINT is negative.

Samples:

Implementation language ST:

```

VAR
  nNumber1 : INT;
  nNumber2 : INT;
  fNumber3 : REAL := 123.456;
  bTruth   : BOOL;
  sText1   : STRING;
  sText2   : STRING := 'Hello World!';
  wsText   : WSTRING;
  dEvent   : DATE := D#2019-9-3;
  nEvent   : UINT;
  nData    : __UXINT;
END_VAR

nNumber1 := TO_INT(4.22);           // Result: 4
nNumber2 := TO_INT(fNumber1);      // Result: 123
bTruth   := TO_BOOL(1);            // Result: TRUE
sText1   := TO_STRING(342);        // Result: '342'
wsText   := TO_WSTRING(sText2);    // Result: "Hello World!"
nEvent   := TO_UINT(dEvent);       // Result: 44288
dEvent   := TO__UXINT(nNumber2);   // Result: 123

```

See also:

- [Data types \[► 756\]](#)

16.3.9.2 <type>_TO_BOOL

This IEC operator converts from a different data type to the data type BOOL.

Syntax: <Data type>_TO_BOOL

The result is TRUE if the operand is not equal to 0. The result is FALSE if the operand equals 0. With the data type STRING the result is TRUE if the operand is TRUE, otherwise the result is FALSE.

Samples:

ST code	Result
bVar := BYTE_TO_BOOL(2#11010101);	TRUE
bVar := INT_TO_BOOL(0);	FALSE
bVar := TIME_TO_BOOL(T#5ms);	TRUE
bVar := STRING_TO_BOOL('TRUE');	TRUE

FBD code	Result
	TRUE
	FALSE
	TRUE
	TRUE

See also:

- Data types > [BOOL](#) [▶ 758]

16.3.9.3 BOOL_TO_<type>

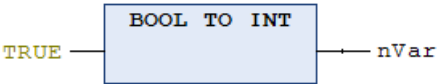
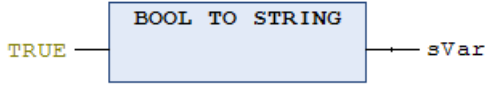
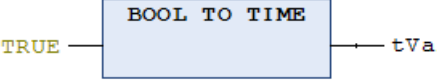
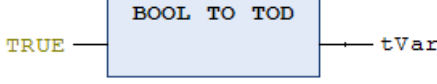
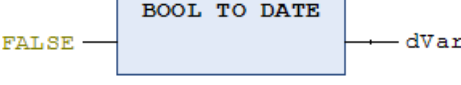
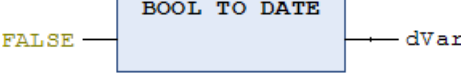
This IEC operator converts from the data type BOOL to a different data type.

Syntax: BOOL_TO_<data type>

With numerical data types the result is 1 if the operand is TRUE and 0 if the operand is FALSE. With the data type STRING the result is TRUE or FALSE.

Samples:

ST code	Result
nVar := BOOL_TO_INT(TRUE);	1
sVar := BOOL_TO_STRING(TRUE);	'TRUE'
tVar := BOOL_TO_TIME(TRUE);	T#1ms
tVar := BOOL_TO_TOD(TRUE);	TOD#00:00:00.001
dVar := BOOL_TO_DATE(FALSE);	D#1970
dtVar := BOOL_TO_DT(TRUE);	DT#1970-01-01-00:00:01

FBD code	Result
	1
	'TRUE'
	T#1ms
	TOD#00:00:00.001
	D#1970-01-01
	DT#1970-01-01-00:00:01

See also:

- Data types > [BOOL](#) [► 758]

16.3.9.4 <INT type>_TO_<INT type>

This IEC operator converts from one integer data type to another integer data type.

Syntax: <INT data type>_TO_<INT data type>

Integral data types:

- BYTE
- WORD, DWORD, LWORD
- SINT, INT, DINT, LINT
- USINT, UINT, UDINT, ULINT



Information may be lost during type conversion from larger to smaller data types. If the number to be converted exceeds the range bound, TwinCAT ignores the first bytes of the number.

Samples:

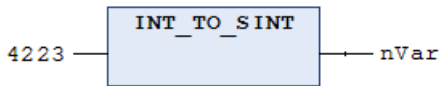
Result: nVar is 127.

ST:

```
nVar := INT_TO_SINT(4223);
```

If you save the integer number 4223 (16#107f in hexadecimal representation) in a SINT variable, this variable contains the number 127 (16#7f in hexadecimal representation).

FBD:



See also:

- Data types > [Integer Data Types \[▶ 758\]](#)

16.3.9.5 REAL/LREAL_TO_<type>

This IEC operator converts from the data types REAL and LREAL to a different data type.

Syntax:

REAL_TO_<data type>

LREAL_TO_<data type>

TwinCAT rounds the Real value of the operands up or down to the nearest integer value and converts to the corresponding data type. The exceptions to this are the data types STRING, BOOL, REAL and LREAL.

i If you convert a number of type REAL or LREAL to SINT, USINT, INT, UINT, DINT, UDINT, LINT or ULINT and the value of the REAL/LREAL number is outside the integer's range of values, you get an undefined result. An exception error is also possible!

When converting to the data type STRING, make sure that the total number of decimal places is limited to 16. If the (L)REAL number contains more digits, the sixteenth digit is rounded and then displayed in the string. If the string length defined for the number is too short, TwinCAT truncates it from the right.

i Information may be lost during type conversion from larger to smaller data types.

Samples:

ST code	Result
nVar1 := REAL_TO_INT(1.5);	2
nVar2 := REAL_TO_INT(1.4);	1
nVar1 := REAL_TO_INT(-1.5);	-2
nVar2 := REAL_TO_INT(-1.4);	-1

FBD code	Result
	2

See also:

- Data types > [REAL/LREAL \[▶ 759\]](#)

16.3.9.6 STRING_TO_<type>

This IEC operator converts from the data type STRING to a different data type.

Syntax: STRING_TO_<data type>

You must specify the operand of type STRING in accordance with the IEC 61131-3 standard. The value must be equivalent to a valid constant (literal) of the target data type. This applies to exponential values, infinite values, prefixes, grouping characters (" ") and commas. Additional characters are allowed behind the digits of a number, e.g. 23xy. Additional characters before a number are not permitted.

The operand must represent a valid value of the target data type.



If the data type of the operand does not match the destination data type or the value is outside the range of the destination data type, the result output is undefined. Information may be lost during type conversion from larger to smaller data types.

Samples:

ST code	Result
bVar := STRING_TO_BOOL('TRUE');	TRUE
nVar := STRING_TO_WORD('abc34');	0
nVar := STRING_TO_WORD('34abc');	34
tVar := STRING_TO_TIME('T#127ms');	T#127ms
fVar := STRING_TO_REAL('1.234');	1,234
nVar := STRING_TO_BYTE('500');	244

FBD code	Result
	TRUE

See also:

- Data types > [STRING](#) [▶ 760]

16.3.9.7 TO_STRING/TO_WSTRING for enumeration variables

If you wish to query the textual identifier of an enumeration component, for example in order to process it further in a text output, add the [Attribute 'to_string'](#) [▶ 836] above the declaration of the enumeration. In the implementation part you can then apply the conversion function TO_STRING or TO_WSTRING to an enumeration variable or to the components of the enumeration. The name of the enumeration component will then be returned.



Attribute 'to_string'

Available from TC3.1 Build 4024

The conversion functions TO_STRING/TO_WSTRING can also be applied to enumerations that are not declared with the attribute 'to_string'. In this case the numerical value of the enumeration component will be returned.

Sample:

Enumeration E_Sample

```
{attribute 'qualified_only'}
{attribute 'strict'}
{attribute 'to_string'}
TYPE E_Sample :
(
  eInit := 0,
  eStart,
  eStop
);
END_TYPE
```

Program MAIN

```
PROGRAM MAIN
VAR
  eSample          : E_Sample;
  nCurrentValue   : INT;
  sCurrentValue   : STRING;
  wsCurrentValue  : WSTRING;
```

```

    sComponent      : STRING;
    wsComponent     : WSTRING;
END_VAR

nCurrentValue := eSample;
sCurrentValue := TO_STRING(eSample);
wsCurrentValue := TO_WSTRING(eSample);

sComponent := TO_STRING(E_Sample.eStart);
wsComponent := TO_WSTRING(E_Sample.eStop);

```

Result of the assignments/conversion functions:

- Value of nCurrentValue: 0
- Value of sCurrentValue: 'eInit'
- Value of wsCurrentValue: "eInit"
- Value of sComponent: 'eStart'
- Value of wsComponent: "eStop"

Result if the enumeration were not to be declared with the attribute 'to_string':

- Value of nCurrentValue: 0
- Value of sCurrentValue: '0'
- Value of wsCurrentValue: "0"
- Value of sComponent: '1'
- Value of wsComponent: "2"

See also:

- [Enumerations \[► 781\]](#)

16.3.9.8 TIME/TOD_TO_<type>

This IEC operator converts from the data type TIME or TIME_OF_DAY (TOD) to a different data type.

Syntax: <TIME data type>_TO_<data type>

TwinCAT internally stores the time in a DWORD in milliseconds (for TIME_OF_DAY since 00:00). TwinCAT converts this value. With the data type STRING the result is the time constant.



Information may be lost during type conversion from larger to smaller data types.

Samples:

ST code	Result
sVar := TIME_TO_STRING(T#12ms);	T#12ms
nVar := TIME_TO_DWORD(T#5m);	300000
nVar := TOD_TO_SINT(TOD#00:00:00.012);	12

FBD code	Result
	T#12ms
	300000
	12

See also:

- Data types > [Date and time data types \[► 761\]](#)

16.3.9.9 DATE/DT_TO_<type>

This IEC operator converts from the data type DATE or DATE_AND_TIME (DT) to a different data type.

Syntax: <DATE data type>_TO_<data type>

TwinCAT saves the date internally in a DWORD in seconds since 1 January 1970. TwinCAT converts this value. With the data type STRING the result is the date constant.



Information may be lost during type conversion from larger to smaller data types.

Samples:

ST code	Result
bVar := DATE_TO_BOOL(D#1970-01-01);	FALSE
nVar := DATE_TO_INT(D#1970-01-15);	29952
nVar := DT_TO_BYTE(DT#1970-01-15-05:05:05);	129
sVar := DT_TO_STRING(DT#1998-02-13-14:20);	'DT#1998-02-13-14:20:00'

FBD code	Result
	FALSE
	29952
	129
	'DT#1998-02-13-14:20:00'

See also:

- Data types > [Date and time data types \[► 761\]](#)

16.3.9.10 TRUNC

This IEC operator converts from the data type REAL to the data type DINT. TwinCAT only uses the integer part of the number.



In TwinCAT 2.x PLC Control, the TRUNC operator converts from REAL to INT. If you import a TwinCAT 2.x PLC project, TwinCAT automatically replaces TRUNC with TRUNC_INT.

If TwinCAT cannot represent the input value with a DINT or INT number, the result of this function is undefined.



For a type conversion operator, if the operand value is outside the value range of the target data type, the result output is undefined. This is the case, for example, when a negative operand value is converted from LREAL to the target data type UINT.

Information may be lost during type conversion from larger to smaller types.

Samples:

Result: nVar1 is 1.

ST:

```
nVar1 := TRUNC(1.9); (* Result: 1 *)  
nVar2 := TRUNC(-1.4); (* Result: -1 *)
```

See also:

- [Data types \[► 756\]](#)

16.3.9.11 TRUNC_INT

The IEC operator is used for conversion from data type REAL to data type INT. TwinCAT only uses the amount of the integer part of the number.



TRUNC_INT corresponds to the operator TRUNC in TwinCAT 2.x PLC and automatically used instead of it when TwinCAT 2.x PLC projects are imported. Note the changed function of TRUNC.

If TwinCAT cannot represent the input value with a DINT or INT number, the result of this function is undefined.



For a type conversion operator, if the operand value is outside the value range of the target data type, the result output is undefined. This is the case, for example, when a negative operand value is converted from LREAL to the target data type UINT.

Information may be lost during type conversion from larger to smaller types.

Samples:

Result: nVar1 is 1.

ST:

```
nVar1 := TRUNC_INT(1.9); (* Result: 1 *)  
nVar2 := TRUNC_INT(-1.4); (* Result: -1 *)
```

See also:

- [Data types \[► 756\]](#)

16.3.10 Comparison operators

The comparison operators are Boolean operators, which compare two inputs (first and second operand).

16.3.10.1 EQ

The IEC operator is used for the "equality" function.

Allowed data types of the operands: any basic data type.

If the operands are the same, the operator returns TRUE, otherwise FALSE.

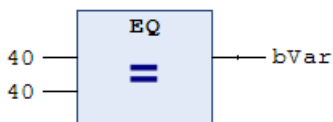
Samples:

Result: bVar is TRUE.

ST:

```
bVar := 40 = 40;
```

FBD:



16.3.10.2 GE

The IEC operator is used for the function "greater than or equal to".

Permitted operand data types: any basic data type.

If the first operand is greater than the second operand or the same size as the second operand, the operator returns TRUE, otherwise FALSE.

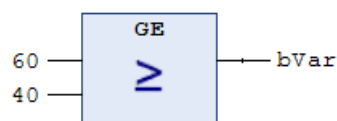
Examples:

Result: bVar is TRUE.

ST:

```
bVar := 60 >= 40;
```

FBD:



16.3.10.3 GT

The IEC operator is used for the "greater than" function.

Permitted operand data types: any basic data type.

If the first operand is greater than the second operand, the operator returns TRUE, otherwise FALSE.

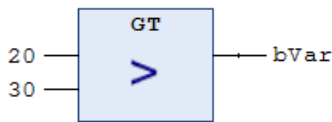
Examples:

Result: bVar is FALSE.

ST:

```
bVar := 20 > 30;
```

FBD:



16.3.10.4 LE

The IEC operator is used for the function "less than or equal to".

Permitted operand data types: any basic data type.

If the first operand is less than the second operand or the same size as the second operand, the operator returns TRUE, otherwise FALSE.

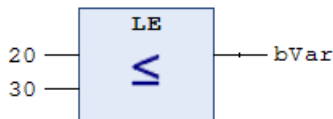
Examples:

Result: bVar is TRUE.

ST:

```
bVar := 20 <= 30;
```

FBD:



16.3.10.5 LT

The IEC operator is used for the "less than" function.

Permitted operand data types: any basic data type.

If the first operand is less than the second operand, the operator returns TRUE, otherwise FALSE.

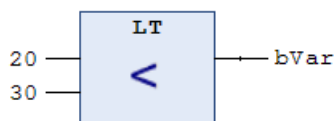
Examples:

Result: bVar is TRUE.

ST:

```
bVar := 20 < 30;
```

FBD:



16.3.10.6 NE

The IEC operator is used for the "inequality" function.

Allowed data types of the operands: any basic data type.

If the operands are not the same, the operator returns TRUE, otherwise FALSE.

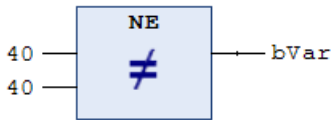
Samples:

Result: bVar is FALSE.

ST:

```
bVar := 40 <> 40;
```

FBD:



16.3.11 Further operators

16.3.11.1 __NEW

The operator is an extension of the IEC 61131-3 standard.

The `__NEW` operator dynamically reserves memory around function blocks to instantiate user-defined data types or arrays of standard data types. The operator returns a suitable typed pointer.

Syntax:

```
<pointer name> := __NEW( <type> ( , <size> )? );
__DELETE( <pointer name> );

<type> : <function block> | <data unit type> | <standard data type>
// (...)?: Optional
```

Sample:

```
pScalarType := __NEW(ScalarType, length);
```

The `__NEW` operator creates an instance of the `<type>` type and returns a pointer to this instance. Then, the initialization of the instance is called. If `<type>` is a standard scalar data type, the optional operand `<size>` is also evaluated. In this case the operator generates an array of type `<standard data type>` and length `<size>` (number of elements). If the memory allocation attempt fails, `__NEW` returns the value 0.

Use the `__NEW` operator within an assignment ("`:=`"), otherwise an error message will be issued.

● No type change is possible via online change

i A function block or user-defined data type whose instance is created dynamically with `__NEW` occupies a fixed memory area. It cannot change its data layout using the online change feature. This means that no new variables can be added, no variables can be deleted, and no types can be changed. This ensures that the pointer to this object remains valid after the online change.

For this reason, the `__NEW` operator can only be applied to function blocks/DUTs from libraries and to function blocks/DUTs with the attribute 'enable dynamic creation' [▶ 801]. If the interface of such a function block/DUT is changed, TwinCAT issues an error message.

Dynamic memory is allocated from the router memory pool.

● Status information of the TwinCAT router

i The function block `FB_GetRouterStatusInfo` from the `Tc2_Uilities` library can be used to read status information of the TwinCAT router, such as the available router memory, from the PLC.

A dynamic memory reserved with `__NEW` must be explicitly released using `__DELETE`. This must be done either already in the same task cycle or at the latest before the PLC is shut down. Otherwise, a so-called memory leak may occur. As a result, other program parts can no longer reserve memory, which can lead to an unstable system. It is recommended to use the PLC library `Tc3_DynamicMemory` to simplify the handling with dynamic memory.

Allocating a STRING variable:

If you wish to allocate a `STRING` of the length `cLength`, use `BYTE` instead of `STRING` as the data type.

```
VAR CONSTANT
    cLength : UINT := 150;
END_VAR
VAR
    bNew    : BOOL;
```



```

    pString : POINTER TO STRING(cLength);
    bDelete : BOOL;
END_VAR

IF bNew AND (pSTRING = 0) THEN
    pString := __NEW(BYTE, cLength);
    bNew := FALSE;
END_IF

IF bDelete THEN
    __DELETE(pSTRING);
    bDelete := FALSE;
END_IF

```

Sample with structure:**Structure ST_sample:**

```

{attribute 'enable_dynamic_creation'}
TYPE ST_Sample :
STRUCT
    a,b,c,d,e,f : INT;
END_STRUCT
END_TYPE

```

MAIN program:

```

PROGRAM MAIN
VAR
    pDut    : POINTER TO ST_Sample;
    bNew    : BOOL := TRUE;
    bDelete : BOOL;
END_VAR

IF bNew AND (pDut = 0) THEN
    pDut := __NEW(ST_Sample);
    bNew := FALSE;
END_IF

IF bDelete THEN
    __DELETE(pDut);
    bDelete := FALSE;
END_IF

```

Sample with function block:**Function block FB_Dynamic:**

```

{attribute 'enable_dynamic_creation'}
FUNCTION_BLOCK FB_Dynamic
VAR
    nCounts : INT;
END_VAR

```

Method Increase:

```

METHOD Increase : INT
VAR_INPUT
    nCountStep : INT;
END_VAR

nCounts := nCounts + nCountStep;
Increase := nCounts;

```

MAIN program:

```

PROGRAM MAIN
VAR
    pFB      : POINTER TO FB_Dynamic;
    nResult  : INT;
    nIncrease : INT := 5;
    bNew     : BOOL;
    bDelete  : BOOL;
END_VAR

IF bNew AND (pFB = 0) THEN
    pFB := __NEW(FB_Dynamic);
    bNew := FALSE;
END_IF

IF (pFB <> 0) THEN
    nResult := pFB^.Increase(nCountStep := nIncrease);

```

```

END_IF

IF bDelete THEN
  __DELETE(pFB);
  bDelete := FALSE;
END_IF

```

Sample with array:

MAIN program:

```

PROGRAM MAIN
VAR
  bNew      : BOOL := TRUE;
  bDelete   : BOOL;
  pArrayBytes : POINTER TO BYTE;
  nTest     : INT;
END_VAR

IF bNew AND (pArrayBytes = 0) THEN
  pArrayBytes := __NEW(BYTE, 25);
  bNew := FALSE;
END_IF

IF (pArrayBytes <> 0) THEN
  pArrayBytes[24] := 125; // writing a value to the last array element
  nTest := pArrayBytes[24]
END_IF

IF bDelete THEN
  __DELETE(pArrayBytes);
  bDelete := FALSE;
END_IF

```

16.3.11.2 __DELETE

The operator is an extension of the IEC 61131-3 standard.

The operator releases the memory of instances, which the operator __NEW generated dynamically. The operator DELETE has no return value, and the operand is set to 0 after this operation.

Syntax: __DELETE (<Pointer>)

If a pointer points to a function block, TwinCAT calls the corresponding method FB_exit before the pointer is set to 0.

● Dealing with dynamic memory

- i** Always specify the exact type of the instance whose memory is to be released. For inheritance, specify the derived function block (variable of type POINTER TO FB_Sub), not the basic function block (variable of type POINTER TO FB_Base).
If the basic function block does not implement an FB_exit function, FB_exit is not called if __DELETE (pfbBase) is called later!

Sample:

Function block FB_Dynamic:

```

FUNCTION_BLOCK FB_Dynamic
VAR_INPUT
  nIn1, nIn2 : INT;
END_VAR
VAR_OUTPUT
  nOut : INT;
END_VAR
VAR
  nTest1 : INT := 1234;
  _inc   : INT := 0;
  _dut   : POINTER TO DUT;
  bNeu   : BOOL;
END_VAR

nOut := nIn1 + nIn2;

```

Method FB_exit:

```
METHOD FB_exit : BOOL
VAR_INPUT
    bInCopyCode : BOOL;
END_VAR
__DELETE(_dut);
```

Method FB_init:

```
METHOD FB_init : BOOL
VAR_INPUT
    bInitRetains : BOOL;
    bInCopyCode : BOOL;
END_VAR
_dut := __NEW(DUT);
```

Method INC:

```
METHOD INC : INT
VAR_INPUT
END_VAR
_inc := _inc + 1;
INC := _inc;
```

Program MAIN:

```
PROGRAM MAIN
VAR
    pFB : POINTER TO FB_Dynamic;
    bInit : BOOL := TRUE;
    bDelete : BOOL;
    nLoc : INT;
END_VAR

IF (bInit) THEN
    pFB := __NEW(FB_Dynamic);
    bInit := FALSE;
END_IF

IF (pFB <> 0) THEN
    pFB^(nIn1 := 1, nIn2 := nLoc, nOut => nLoc);
    pFB^.INC();
END_IF

IF (bDelete) THEN
    __DELETE(pFB);
END_IF
```

16.3.11.3 __ISVALIDREF

The operator is an extension of the IEC 61131-3 standard.

The operator is used to check whether a reference points to a value. The check is thus comparable with a check for 'unequal to 0' in the case of a pointer variable.

A description of the application and a sample of the use of the operator can be found in the description of the data type [REFERENCE](#) [► 770].

i Checking interface variables

The operator `__ISVALIDREF` can only be used for operands of type `REFERENCE TO`. This operator cannot be used for checking interface variables. To check whether an interface variable was already assigned a function block instance, you can check the interface variable for not equal to 0 (`IF iSample <> 0 THEN ...`).

16.3.11.4 __QUERYINTERFACE

The operator is an extension of the IEC 61131-3 standard.

At runtime, the operator performs a type conversion of one interface reference to another. The operator returns a result of type `BOOL`. `TRUE` means that TwinCAT performed the conversion successfully.

Syntax: `__QUERYINTERFACE(<ITF_Source>, <ITF_Dest>);`

1. Operand: Interface variable or FB instance
2. Operand: Interface variable with desired target types

A prerequisite for explicit conversion is that both ITF_Source and ITF_Dest are derivatives of `__System.IQueryInterface`. This interface is implicitly available and requires no library.

Sample:

Interfaces:

```
INTERFACE I_Base EXTENDS __System.IQueryInterface
METHOD BaseMethod : BOOL

INTERFACE I_Sub1 EXTENDS I_Base
METHOD SubMethod1 : BOOL

INTERFACE I_Sub2 EXTENDS I_Base
METHOD SubMethod2 : BOOL

INTERFACE I_Sample EXTENDS __System.IQueryInterface
METHOD SampleMethod : BOOL
```

Function blocks:

```
FUNCTION_BLOCK FB_1 IMPLEMENTS I_Sub1
METHOD BaseMethod : BOOL
  BaseMethod := TRUE;
METHOD SubMethod1 : BOOL
  SubMethod1 := TRUE;

FUNCTION_BLOCK FB_2 IMPLEMENTS I_Sub2
METHOD BaseMethod : BOOL
  BaseMethod := FALSE;
METHOD SubMethod2 : BOOL
  SubMethod2 := TRUE;

FUNCTION_BLOCK FB_3 IMPLEMENTS I_Base, I_Sample
METHOD BaseMethod : BOOL
  BaseMethod := FALSE;
METHOD SampleMethod : BOOL
  SampleMethod := FALSE;
```

Program:

```
PROGRAM MAIN
VAR
  fb1      : FB_1;
  fb2      : FB_2;
  fb3      : FB_3;
  iBase1   : I_Base := fb1;
  iBase2   : I_Base := fb2;
  iBase3   : I_Base := fb3;
  iSub1    : I_Sub1 := 0;
  iSub2    : I_Sub2 := 0;
  iSample  : I_Sample := 0;
  bResult1 : BOOL;
  bResult2 : BOOL;
  bResult3 : BOOL;
  bResult4 : BOOL;
  bResult5 : BOOL;
END_VAR

// Result: bResult1 = TRUE as the conversion is successful => iSub1 references fb1
// Explanation: iBase1 references the object fb1 of type FB_1 that implements the interface I_Sub1
bResult1 := __QUERYINTERFACE(iBase1, iSub1);

// Result: bResult2 = FALSE as the conversion is not successful => iSub2 = 0
// Explanation: iBase1 references the object fb1 of type FB_1 that does not implement the interface
// I_Sub2
bResult2 := __QUERYINTERFACE(iBase1, iSub2);

// Result: bResult3 = FALSE as the conversion is not successful => iSub1 = 0
// Explanation: iBase2 references the object fb2 of type FB_2 that does not implement the interface
// I_Sub1
bResult3 := __QUERYINTERFACE(iBase2, iSub1);

// Result: bResult4 = TRUE as the conversion is successful => iSub2 references fb2
// Explanation: iBase2 references the object fb2 of type FB_2 that implements the interface I_Sub2
bResult4 := __QUERYINTERFACE(iBase2, iSub2);

// Result: bResult5 = TRUE as the conversion is successful => iSample references fb3
```

```
// Explanation: iBase3 references the object fb3 of type FB_3 that implements the interface I_Sample
bResult5 := __QUERYINTERFACE(iBase3, iSample);
```

16.3.11.5 __QUERYPOINTER

The operator is an extension of IEC61131-3.

The operator enables type conversion of an interface reference of a function block to a pointer at runtime. The operator returns a result of type BOOL. TRUE means that TwinCAT performed the conversion successfully.



For compatibility reasons, the definition of the pointer to be converted must be an extension of the basic interface __SYSTEM.IQueryInterface.

Syntax: __QUERYPOINTER (<ITF_Source>, <Pointer_Dest>)

The first operand assigned to the operator is an interface reference or an FB instance with the desired target types, the second operand is a pointer. After __QUERYPOINTER has been processed, Pointer_Dest contains the pointer to the reference or instance of a function block, to which the interface reference ITF_Source currently points. Pointer_Dest is not typed and can be cast to any type. Make sure the type is correct. For example, the interface could offer a method that returns a type code.

Example:

Interfaces:

```
INTERFACE I_Base EXTENDS __System.IQueryInterface
METHOD Base : BOOL

INTERFACE I_Derived EXTENDS I_Base
METHOD Derived : BOOL
```

Function block:

```
FUNCTION_BLOCK FB_Variante IMPLEMENTS I_Derived
METHOD Base : BOOL
METHOD Derived : BOOL
```

Program:

```
PROGRAM MAIN
VAR
  iDerived : I_Derived;
  fbVariante : FB_Variante;
  bResult : BOOL;
  bTest : BOOL;
  pFB : POINTER TO FB_Variante;
END_VAR

iDerived := fbVariante;
bResult := __QUERYPOINTER(iDerived, pFB);

IF bResult THEN
  bTest := pFB^.Derived();
END_IF
```

16.3.11.6 __TRY, __CATCH, __FINALLY, __ENDTRY

The operators are an extension of the IEC 61131-3 standard and are used for a targeted exception handling in the IEC code.



Available from TC3.1 Build 4024 for 32-bit runtime systems

Available from TC3.1 Build 4026 for 64-bit runtime systems

Syntax:

```

__TRY
  <try_statements>

__CATCH (exc)
  <catch_statements>

__FINALLY
  <finally_statements>

__ENDTRY
<further_statements>

```

If an instruction that appears under the operator `__Try` generates an exception, the PLC program does not stop. Instead, it executes the instructions under `_CATCH` and thus starts the exception handling. The instructions under `__FINALLY` are then executed. The exception handling ends with `__ENDTRY`. The PLC program then executes the subsequent instructions (instructions after `__ENDTRY`).

The instructions of the `_TRY` block, which are located under the instruction that triggers the exception, are no longer executed. This means that as soon as the exception is discarded, the further execution of the `_TRY` block is aborted and the instructions under `_CATCH` are executed.

The instructions under `_FINALLY` are always executed, i.e. even if the instructions under `_TRY` do not throw any exception.

An IEC variable for an exception has the Data type `SYSTEM.ExceptionCode` [[▶ 740](#)].

NOTICE

Machine downtime due to intercepted floating point exceptions on x86 target systems

Due to a technical limitation of the x86 platform, intercepted exceptions caused by a floating point operation can have unintended, serious consequences:

The system may be put into an unrecoverable state or a machine downtime may occur due to a stack overflow. It can also happen that the machine does not come to a standstill immediately, but later during further operations that are also carried out on the stack.

- Use implicit check functions when floating point operations are used within try-catch blocks on x86 targets.

Sample

The `_TRY` block in the following sample contains a pointer access and a division. In the first cycle an "Access Violation" exception is thrown inside this block, because the pointer `pSample` is still a NULL-pointer at this point in time. The use of a NULL-pointer leads to an exception. In the second cycle a further exception is thrown inside the `_TRY` block - this time it is an exception due to a division by 0.

Both exception causes are resolved within the `__CATCH` statements: the first exception by a value correction of the pointer `pSample` and the second exception by a value correction of the divisor `nDivisor`.

The fact that the exception handling works and how it works can be comprehended at runtime as follows:

- Since the erroneous accesses that normally lead to a runtime exception and to a corresponding stopping of the program execution are intercepted within the exception handling, the TC runtime remains in Run mode and the program execution continues.
- Since two exceptions are intercepted, the counter `nCounter_CATCH` is incremented twice and thus has the value 2.
- Since the instructions described below are executed in each cycle, the counters incremented there all have the same value. The value corresponds to the number of cycles so far.
 - the instructions before the TRY-CATCH block = incrementation of the counter `nCounter1`
 - the instructions at the start of the `_TRY` block = incrementation of the counter `nCounter_TRY1`
 - the instructions under `_FINALLY` = incrementation of the counter `nCounter_FINALLY`
 - the instructions after the TRY-CATCH block = incrementation of the counter `nCounter2`
- Since the execution of the `_TRY` block is aborted twice due to the two thrown exceptions, the end of the `_TRY` instructions is not reached in two cycles. The variable value of `nCounter_TRY2` is thus smaller than that of `nCounter_TRY1` by 2.

- Since the causes of the two exceptions are remedied inside the `_CATCH` instructions (assign a valid address to `pSample` and a value other than 0 to `nDivisor`), each of the two exceptions occurs only once.
- The two exceptions are saved in the sample in a global array for creating an exception history. Following the two exceptions, the corresponding index variable `nExcIndex` is thus 2 and the array `aExceptionHistory` has the following values:
 - `aExceptionHistory[0]` = `RTSEXCPT_ACCESS_VIOLATION`
 - `aExceptionHistory[1]` = `RTSEXCPT_DIVIDEBYZERO`
 - `aExceptionHistory[2]` to `aExceptionHistory[10]` = `RTSEXCPT_NOEXCEPTION`
- Following the two exceptions the variable `exc` has returned to the default value `RTSEXCPT_NOEXCEPTION` and the variable `lastExc` indicates the last occurring exception code with `RTSEXCPT_DIVIDEBYZERO`.

Global variable list "GVL_Exc":

```
{attribute 'qualified_only'}
VAR_GLOBAL CONSTANT
  cMaxExc          : UINT := 10;
END_VAR
VAR_GLOBAL
  nExcIndex        : UINT;
  aExceptionHistory : ARRAY[0..cMaxExc] OF __SYSTEM.ExceptionCode;
END_VAR
```

Function "F_SaveExceptionCode":

```
FUNCTION F_SaveExceptionCode
VAR_INPUT
  excInput          : __SYSTEM.ExceptionCode;
END_VAR

// Log the thrown exception into the global exception history array
IF GVL_Exc.nExcIndex <= GVL_Exc.cMaxExc THEN
  GVL_Exc.aExceptionHistory[GVL_Exc.nExcIndex] := excInput;
  GVL_Exc.nExcIndex := GVL_Exc.nExcIndex + 1;
END_IF
```

Program "MAIN":

```
PROGRAM MAIN
VAR
  nCounter1          : INT;
  nCounter2          : INT;
  nCounter_TRY1      : INT;
  nCounter_TRY2      : INT;
  nCounter_CATCH     : INT;
  nCounter_FINALLY   : INT;

  exc                : __SYSTEM.ExceptionCode;
  lastExc            : __SYSTEM.ExceptionCode;

  pSample            : POINTER TO BOOL;
  bVar               : BOOL;
  nSample            : INT := 100;
  nDivisor           : INT;
END_VAR

// Counter 1
nCounter1 := nCounter1 + 1;

// TRY-CATCH block
__TRY
  nCounter_TRY1 := nCounter_TRY1 + 1;
  pSample^      := TRUE;           // 1. cycle: null pointer access leads to "access
violation" exception
  nSample       := nSample/nDivisor; // 2. cycle: division by zero leads to "divide by zero"
exception
  nCounter_TRY2 := nCounter_TRY2 + 1;

__CATCH(exc)
  nCounter_CATCH := nCounter_CATCH + 1;

  // Exception logging
  lastExc := exc;
  F_SaveExceptionCode(excInput := exc);
```

```

// Correct the faulty variable values
IF (exc = __SYSTEM.ExceptionCode.RTSEXCPT_ACCESS_VIOLATION) AND (pSample = 0) THEN
    pSample := ADR(bVar);
ELSIF ((exc = __SYSTEM.ExceptionCode.RTSEXCPT_DIVIDEBYZERO) OR (exc =
__SYSTEM.ExceptionCode.RTSEXCPT_FPU_DIVIDEBYZERO)) AND (nDivisor = 0) THEN
    nDivisor := 1;
END_IF

__FINALLY
    nCounter_FINALLY := nCounter_FINALLY + 1;

__ENDTRY

// Counter 2
nCounter2 := nCounter2 + 1;

```

16.3.11.6.1 Data type __SYSTEM.ExceptionCode

See also: [TRY, CATCH, FINALLY, ENDTRY](#) | [737](#)

```

TYPE ExceptionCode :
(
    RTSEXCPT_UNKNOWN                := 16#FFFFFFF,
    RTSEXCPT_NOEXCEPTION            := 16#00000000,
    RTSEXCPT_WATCHDOG               := 16#00000010,
    RTSEXCPT_HARDWAREWATCHDOG       := 16#00000011,
    RTSEXCPT_IO_CONFIG_ERROR        := 16#00000012,
    RTSEXCPT_PROGRAMCHECKSUM        := 16#00000013,
    RTSEXCPT_FIELDBUS_ERROR         := 16#00000014,
    RTSEXCPT_IOUPDATE_ERROR         := 16#00000015,
    RTSEXCPT_CYCLE_TIME_EXCEED      := 16#00000016,
    RTSEXCPT_ONLCHANGE_PROGRAM_EXCEEDED := 16#00000017,
    RTSEXCPT_UNRESOLVED_EXTREFS     := 16#00000018,
    RTSEXCPT_DOWNLOAD_REJECTED      := 16#00000019,
    RTSEXCPT_BOOTPROJECT_REJECTED_DUE_RETAIN_ERROR := 16#0000001A,
    RTSEXCPT_LOADBOOTPROJECT_FAILED := 16#0000001B,
    RTSEXCPT_OUT_OF_MEMORY           := 16#0000001C,
    RTSEXCPT_RETAIN_MEMORY_ERROR     := 16#0000001D,
    RTSEXCPT_BOOTPROJECT_CRASH       := 16#0000001E,
    RTSEXCPT_BOOTPROJECTTARGETMISMATCH := 16#00000021,
    RTSEXCPT_SCHEDULEERROR           := 16#00000022,
    RTSEXCPT_FILE_CHECKSUM_ERR       := 16#00000023,
    RTSEXCPT_RETAIN_IDENTITY_MISMATCH := 16#00000024,
    RTSEXCPT_IEC_TASK_CONFIG_ERROR   := 16#00000025,
    RTSEXCPT_APP_TARGET_MISMATCH     := 16#00000026,
    RTSEXCPT_ILLEGAL_INSTRUCTION     := 16#00000050,
    RTSEXCPT_ACCESS_VIOLATION        := 16#00000051,
    RTSEXCPT_PRIV_INSTRUCTION        := 16#00000052,
    RTSEXCPT_IN_PAGE_ERROR           := 16#00000053,
    RTSEXCPT_STACK_OVERFLOW          := 16#00000054,
    RTSEXCPT_INVALID_DISPOSITION     := 16#00000055,
    RTSEXCPT_INVALID_HANDLE          := 16#00000056,
    RTSEXCPT_GUARD_PAGE               := 16#00000057,
    RTSEXCPT_DOUBLE_FAULT            := 16#00000058,
    RTSEXCPT_INVALID_OPCODE          := 16#00000059,
    RTSEXCPT_MISALIGNMENT             := 16#00000100,
    RTSEXCPT_ARRAYBOUNDS              := 16#00000101,
    RTSEXCPT_DIVIDEBYZERO             := 16#00000102,
    RTSEXCPT_OVERFLOW                 := 16#00000103,
    RTSEXCPT_NONCONTINUABLE           := 16#00000104,
    RTSEXCPT_PROCESSORLOAD_WATCHDOG   := 16#00000105,
    RTSEXCPT_FPU_ERROR                := 16#00000150,
    RTSEXCPT_FPU_DENORMAL_OPERAND     := 16#00000151,
    RTSEXCPT_FPU_DIVIDEBYZERO         := 16#00000152,
    RTSEXCPT_FPU_INEXACT_RESULT       := 16#00000153,
    RTSEXCPT_FPU_INVALID_OPERATION    := 16#00000154,
    RTSEXCPT_FPU_OVERFLOW             := 16#00000155,
    RTSEXCPT_FPU_STACK_CHECK          := 16#00000156,
    RTSEXCPT_FPU_UNDERFLOW            := 16#00000157,
    RTSEXCPT_VENDOR_EXCEPTION_BASE    := 16#00002000,
    RTSEXCPT_USER_EXCEPTION_BASE      := 16#00010000
) UDINT ;
END_TYPE

```


16.3.11.7 __VARINFO

The operator is an extension of the IEC 61131-3 standard. The operator returns information about a variable. You can save the information as a data structure in a variable of data type __SYSTEM.VAR_INFO.

Syntax in the declaration:

```
<name of the info variable> : __SYSTEM.VAR_INFO; // Data structure for info variable
```

Syntax for the call:

```
<name of the info variable> := __VARINFO( <variable name> ); // Call of the operator
```

Sample:

At runtime the variable MyVarInfo contains the information about the variable nVar.

```
VAR
  MyVarInfo : __SYSTEM.VAR_INFO;
  nVar      : INT;
END_VAR
MyVarInfo := __VARINFO(nVar);
```

Data type SYSTEM.VAR_INFO

A variable of data type __SYSTEM.VAR_INFO contains:

Name	Data type	Initialization	Description
ByteAddress	DWORD	0	Address of the variables Sample: 16#072E35EC Note: During bit access of a variable <variable name>.<bit index>, the address of the variable containing the bit is specified.
ByteOffset	DWORD	0	Offset of the variable address in bytes. Sample: 13936 bytes. Note: If the variable is global, the offset is relative to the start of the area. If the variable is a local variable in a function or method, the offset is relative to the current stack frame. If the variable is a local variable in a function block, the offset is relative to the function block instance.
Area	DINT	0	Memory area number in the runtime system. Sample: -1. This means that the variable is not stored globally in memory, but relative to an instance or on the stack. Note: The memory areas are device-dependent.
BitNo	INT	0	Number of bits in bytes Sample: 16#00FF bytes Note: If the variable is not an integer data type: BitNo = -1 = 16#FFFF
BitSize	INT	0	Memory size of the variables in bits Sample: 16 bits
BitAddress	UDINT	0	Bit address of the variables Requirement: The variable is in input memory area I, output memory area Q or flag memory area M. Otherwise the value is undefined.
TypeClass	TYPE_CLASS	TYPE_BOOL	Data type class of the variables Sample: TYPE_INT, TYPE_ARRAY Note: For user-defined data types or function block instances, the system outputs the data type class TYPE_USERDEF.
TypeName	STRING(79)	''	Data type name of the variable as STRING(79) Note: For user-defined data types, the function block name or DUT name is specified. Sample: 'INT', 'ARRAY'
NumElements	UDINT	0	Number of array elements Requirement: The variable has the data type ARRAY. Sample: 8
BaseTypeClasses	TYPE_CLASS	TYPE_BOOL	Elementary basic data type of the array elements. Requirement: The variable has the data type ARRAY. Sample: TYPE_INT with arrA : ARRAY [1..2,1..2,1..2] OF INT;
ElemBitSize	UDINT	0	Memory size of the array element in bits Requirement: The variable has the data type ARRAY. Sample: 16 bits at arrA : ARRAY [1..2,1..2,1..2] OF INT;

MemoryArea	MEMORY_AR EA	MEM_MEMO RY	Information on the memory area: <ul style="list-style-type: none"> MEM_GLOBAL: Global memory area For example in area 0 MEM_LOCAL: Local memory area in Area -1 MEM_MEMORY: Flag memory area %M For example in 16#10 in area 1 MEM_INPUT: Input memory area %I For example in 16#04 in area 2 MEM_OUTPUT: Output memory area %Q For example in 16#08 in area 3 MEM_RETAIN: Retain memory area For example in 16#20 in area 0 Sample: MEM_GLOBAL
Symbol	STRING(39)	“	Variable name as STRING(39) Sample: 'iCounter', 'arrA'
Comment	STRING(79)	“	Comment of the variable declaration Sample: 'Counts the calls' or 'Stores the A data'

16.3.11.8 __POUNAME



Available from TC3.1 Build 4026.

The operator is an extension of the IEC 61131-1 standard.

The operator returns the name of the program organization unit (POU) containing the __POUNAME operator at runtime. To do this, the operator must be assigned to a variable of type `STRING` in the declaration part or in the implementation part.

The result of __POUNAME depends on where it is used:

- Within a program: program name
- Within a function: function name
- Within a function block: name of the function block
- Within a method: name of the method, qualified with the name of the function block
- Inside a Get/Set accessor in a property: name of the property, qualified with the name of the function block, + Get/Set
- Within a GVL: name of the GVL
- Within a structure: name of the structure
- Within a data structure UNION: name of the UNION

Sample:

```
PROGRAM MAIN
VAR
    sPouNameDecl : STRING := __POUNAME(); //Liefert 'MAIN'
    sPouNameImpl : STRING;
END_VAR
sPouNameImpl:= __POUNAME(); //Liefert 'MAIN'
```

16.3.11.9 __POSITION



Available from TC3.1 Build 4026.

The operator is an extension of the IEC 61131-1 standard.

The operator returns the position of a variable in the declaration part or in the implementation part of a programming block at runtime. For this purpose, the operator `__POSITION` must be assigned to a variable of type `STRING` in the declaration part or in the implementation part.

Result of `__POSITION`:

- **Declaration part:** Line <line number> (Decl)
- **Implementation part:** Line <line number>, Column <Column number> (Impl)

Sample:

```
PROGRAM MAIN
VAR
  sPositionDecl : STRING := __POSITION(); //Liefert die Zeilennummer dieser Deklaration
  sPositionImpl : STRING;
END_VAR
sPositionImpl := __POSITION(); //Liefert Zeilen- und Spaltennummer dieser Zuweisung
```

16.4 Operands

Constants and literals

Constants are identifiers for unchanging values. You can declare constants locally within a programming block or globally within a global variable list. The declaration section is extended with the keyword `CONSTANT` for this purpose.

Constants are also strings that represent the value of a base type such as integers or floating-point numbers, for example `16#FFFF_FFFF`, `T#5s` or `-1.234 E-5`. To distinguish them, such constants are also called literals, literal constants, or unnamed constants. There are logical (`TRUE`, `FALSE`) or numeric literals (`3.1415`, `T#5s`), but also character literals (`'Hello world!'`, `"black"`).

Syntax declaration:

```
<scope> CONSTANT
  <identifier>:<data type> := <initial value>;
END_VAR

<scope>          : VAR | VAR_INPUT | VAR_STAT | VAR_GLOBAL
<data type>     : <elementary data type | user defined data type | function block >
<initial value> : literal value | identifier | expression
```

Allowed initial values:

- Literal, for example `TRUE`, `FALSE`, `16#FFFF_FFFF`
- Named constant declared elsewhere.
- Simple expression from literals, also combined with simple operators such as `+` `-` `*`.

Inputs or function calls cannot be specified as initial values.

Sample:

```
VAR_GLOBAL CONSTANT
  cMax      : INT := 100;
  cSpecial : INT := cMax - 10;
END_VAR
```

Constants are only described in the declaration. The assignment of an initial value is mandatory. Within an implementation, constants are read only and therefore always appear to the right of the allocation operator in an instruction.

The constants are replaced with the initial value when the code is compiled. It must also be possible to calculate the initial value at compile time.

Constants of structured or user-defined types are calculated only at runtime. Structured constants in programs or GVLs are calculated once at program start. Structured constants in functions or methods are calculated each time the function or method is called. The initialization of structured constants can thus depend on inputs or execute function calls.

See also:

- [BOOL constants \[▶ 745\]](#)
- [Number constants \[▶ 745\]](#)
- [REAL/LREAL constants \[▶ 746\]](#)
- [STRING constants \[▶ 747\]](#)
- [TIME/LTIME constants \[▶ 747\]](#)
- [Date and time constants \[▶ 749\]](#)
- [Typed constants / typed literals \[▶ 751\]](#)

Variables

You can declare variables either locally in the declaration part of a function block, or in a global variable list.

At which point you can use a variable depends on its data type.

See also:

- [Accessing variables of arrays, structures and function blocks \[▶ 751\]](#)
- [Bit access to variables \[▶ 752\]](#)




Further

- [Addresses \[▶ 754\]](#)
- [Functions \[▶ 756\]](#)

See also:

- [Declaring variables \[▶ 66\]](#)
- [Using the input wizard \[▶ 135\]](#)
- [Constant variables - CONSTANT \[▶ 689\]](#)
- [ST Expressions \[▶ 625\]](#)

Also see about this

-  [Using the input wizard \[▶ 135\]](#)
-  [Using the input wizard \[▶ 135\]](#)
-  [Declaring variables \[▶ 66\]](#)

16.4.1 BOOL Constants

BOOL constants are the truth values TRUE (1) and FALSE (0).

See also:

- [BOOL \[▶ 758\]](#)

16.4.2 Numeric Constants

Numerical values can be binary numbers, octal numbers, decimal numbers or hexadecimal numbers. If an integer value is not a decimal number, its base must be written before the integer constant, followed by the hash symbol (#). For hexadecimal numbers, the numerals for the numbers 10 to 15 are represented by the letters A-F, as usual.

You can use underscores within a numerical value.

Examples:

14	Decimal number
2#1001_0011	Binary number
8#67	Octal number
16#A	Hexadecimal number
DINT#16#A1	Typed data type DINT# and base 16# combined.

The type of this numerical values can be BYTE, WORD, DWORD, SINT, USINT, INT, UINT, DINT, UDINT, REAL or LREAL.



Implicit conversions from "larger" to "smaller" types are not allowed. You cannot simply use a DINT variable as an INT variable. To do this, you must use a type conversion function.



Since number constants are generally treated as integer values, in divisions the constant has to be specified as a floating-point number, in order to avoid losing the rest. Example: Division 1/10 results in 0, division 1.0/10 results in 0.1.

See also:

- [Operators \[► 696\]](#)
- [Typed constants / typed literals \[► 751\]](#)

16.4.3 REAL/LREAL Constants

You can specify floating-point numbers as REAL and LREAL constants in dot notation or in exponential notation with mantissa and exponent. The dot serves as a decimal symbol according to the International System of Units (English).

Syntax Exponential representation:

```
<significand> e | E <exponent>
```

```
exponent : -44..38 // REAL
exponent : -324..308 // LREAL
```

Examples:

7.4	Decimal number; 7,4 with comma returns compiler error.
1/3.0	Decimal fraction for 0.333333343 Note: When dividing integer types, the result remains an integer type. Truncation takes place. For example, 1/3 returns 0 as the result.
1.64e+009	Exponential representation

REAL literal

-3.402823e+38	Smallest number
-1E-44	Largest negative number
1.0E-44	Smallest positive number
3.402823e+38	Largest number

LREAL literal

-1.7976931348623157E+308	Smallest number
-4.94065645841247E-324	Largest negative number
4.94065645841247E-324	Smallest positive number
1.7976931348623157E+308	Largest number

i Allocation of a particularly large literal

To assign an integer literal that is greater than the upper limit of ULINT, either set a comma or specify an explicit typecast. Without such a specification as a floating-point number, information can be lost.

Example:

```
fMyReal : REAL := 3400000000000000000000.0;
fMyReal : REAL := REAL#3400000000000000000000;
```

See also:

- [REAL/LREAL \[► 759\]](#)

16.4.4 STRING Constants

A STRING constant is a string enclosed by single quotation marks. The characters are encoded according to the Windows 1252 character set. As a subset of Windows-1252, the character set of ISO/IEC 8859-1 is supported. A STRING constant can contain blanks and umlauts, since these characters are part of the character set. It is also referred to as a character literal or simply a string.

Example:

```
'Hello World!'
```

If a STRING constant contains a dollar sign (\$), the following two characters are interpreted as hexadecimal code according to the Windows-1252 coding. The code also corresponds to the ASCII code. Also note the special cases.

Hexadecimal code

String with \$ code	Interpretation
\$<8-bit code>	8-Bit code: two-digit hexadecimal number that is interpreted according to ISO/IEC 8859-1.
'\$41'	A
'\$A9'	©
'\$40'	@
'\$0D'	Control character line break, corresponds to '\$R'.
'\$0A'	Control character new line, corresponds to '\$L' and '\$SN'.

Special cases

String with \$ code	Interpretation
'\$L', '\$l'	Control character line feed, corresponds to '\$0A'.
'\$N', '\$n'	Control character new line, corresponds to '\$0A'.
'\$P', '\$p'	Control character form feed
'\$R', '\$r'	Control character line break, corresponds to '\$0D'.
'\$T', '\$t'	Control character tab
'\$\$'	Dollar sign: \$
'\$'	Single quotation mark: '

Example: constant declaration

```
VAR CONSTANT
  sConstA : STRING := 'Hello Allgäu';
  sConstB : STRING := 'Hello Allgäu $21'; // Hello Allgäu!
END_VAR
```

16.4.5 TIME/LTIME constants

You can use TIME constants to handle the standard timer modules. The constant has a size of 32 bits and thus a resolution in milliseconds.

In addition, the time constant LTIME is available as a time base for high-resolution timers. The LTIME constant has a size of 64 bits and thus a resolution in nanoseconds.

TIME constant

Syntax:

```
<time keyword> # <length of time>
<time keyword> : TIME | time | T | t
<length of time> : ( <number of days>d )? ( <number of hours>h )? ( <number of minutes>m )?
( <number of seconds>s )? ( <number of milliseconds>ms)? // ( ...)? Optional
```

The order of the time units must not be changed. It is not necessary to use all units.

Time units:

- D | d : days
- H | h : hours
- M | m : minutes
- S | s : seconds
- MS | ms : milliseconds

Examples: Correct time constants in an ST assignment

```
VAR
  tLength0 : TIME := T#14ms;
  tLength1 : TIME := T#100s12ms; // Overflow in the highest unit is allowed.
  tLength2 : TIME := T#12h34m15s;
  tCompare : TIME;
  bOK : BOOL;
  tLongest := T#49D17H2M47S295MS; // 4294967295
END_VAR

IF tLength < T#15MS THEN
  IF tCompare < tLength1 THEN
    bOK := TRUE;
  END_IF;
END_IF
```

Examples: Incorrect use of time constants

tIncorrect1 := t#5m68s;	Overflow at a lower position
tIncorrect2 := 15ms;	Time identifier T# missing
tIncorrect3 := t#4ms13d;	Incorrect order of time units

LTIME constant

Syntax:

```
<long time keyword> # <length of high resolution time>
<long time keyword> : LTIME | ltime
<length of high resolution time> : <length of time> ( <number of microseconds>us )? ( <number of
nanoseconds>ns )? // ( ...)? Optional
```

You can use the same time units for LTIME constants as for TIME constants. Additionally, you can specify microseconds and nanoseconds, since the time specification is calculated in higher temporal resolution. Internally LTIME literals are treated like the data type LWORD and therefore the value is resolved in nanoseconds.

Additional time units:

- US | us : microseconds
- NS | ns : nanoseconds

Examples: Correct time constants in an ST assignment

```
VAR
  tLength0 : TIME := LTIME#1000d15h23m12s34ms2us44ns;
  tLength1 : TIME := LTIME#3445343m3424732874823ns;
END_VAR
```


16.4.6 Date and time constants

32-bit dates 'DATE'

Use the DATE (D) keyword to specify dates.

Syntax:

```
<date keyword>#<year>-<month>-<day>
<date keyword> : DATE | date | D | d
<year> : 1970-2106
<month> : 1-12
<day> : 1-31
```

DATE literals are internally treated like the DWORD data type, which corresponds to an upper limit of DATE#2106-2-7.

Example:

```
PROGRAM MAIN
VAR
  dStart : DATE := DATE#2018-8-8;
  dEnd : DATE := D#2018-8-31;
  dCompare : DATE := date#1996-05-06;
  bInTime : BOOL;

  dEarliest : DATE := d#1970-1-1; // = 0
  dLatest : DATE := DATE#2106-2-7; // = 4294967295
END_VAR

IF dStart < dCompare THEN
  IF dCompare < dEnd THEN
    bInTime := TRUE;
  END_IF;
END_IF
```

64-bit dates 'LDATE'

Use the LDATE (LD) keyword to specify dates.

Syntax:

```
<date keyword>#<year>-<month>-<day>
<date keyword> : LDATE | ldate | LD | ld
<year> : 1970-2262
<month> : 1-12
<day> : 1-31
```

LDATE literals are internally treated like the LWORD data type, which corresponds to an upper limit of DATE#2554-7-21.

Example:

```
PROGRAM MAIN
VAR
  dStart : LDATE := LDATE#2018-8-8;
  dEnd : LDATE := ldate#2018-8-31;
  dCompare : LDATE := LD#1996-05-06;
  bInTime : BOOL;

  dEarliest : LDATE := d#1970-1-1; // = 0
  dLatest : LDATE := DATE#2106-2-7; // = 4294967295
END_VAR

IF dStart < dCompare THEN
  IF dCompare < dEnd THEN
    bInTime := TRUE;
  END_IF;
END_IF
```

32-bit date and time specifications 'DATE_AND_TIME'

Use the DATE_AND_TIME (DT) keyword to specify date and time.

Syntax:

<date and time keyword>#<date and time value>

<date and time keyword> : DATE_AND_TIME | date_and_time | DT | dt
 <date and time value> : <year>-<month>-<day>-<hour>:<minute>:<second>
 <year> : 1970-2106
 <month> : 1-12
 <day> : 1-31
 <hour> : 0-24
 <minute> : 0-59
 <second> : 0-59

DATE_AND_TIME literals are internally handled as DWORD data type. The time is processed in seconds and can therefore take values from January 1, 1970 00:00 to February 07, 2106 6:28:15.

Example:

```
PROGRAM MAIN
VAR
  dtDate0   : DATE_AND_TIME := DATE_AND_TIME#1996-05-06-15:36:30;
  dtDate1   : DATE_AND_TIME := DT#1972-03-29-00:00:00;
  dtDate2   : DT             := DT#2018-08-08-13:33:20.5;

  dtEarliest : DATE_AND_TIME := DATE_AND_TIME#1979-1-1-00:00:00; // 0
  dtLatest   : DATE_AND_TIME := DATE_AND_TIME#2106-2-7-6:28:15; // 4294967295
END_VAR
```

64-bit date and time specifications 'LDATE_AND_TIME'

Use the LDATE_AND_TIME (LDT) keyword to specify date and time.

Syntax:

<date and time keyword>#<long date and time value>

<date and time keyword> : LDATE_AND_TIME | ldate_and_time | LDT | ldt
 <date and time value> : <year>-<month>-<day>-<hour>:<minute>:<second>
 <year> : 1970-2554
 <month> : 1-12
 <day> : 1-31
 <hour> : 0-24
 <minute> : 0-59
 <second> : 0-59

LDATE_AND_TIME literals are internally handled as LWORD data type. The time is processed in seconds and can therefore have values from January 1, 1970 00:00 to July 21, 2554 23:59:59.999999999.

Example:

```
PROGRAM MAIN
VAR
  dtDate0   : LDATE_AND_TIME := LDATE_AND_TIME#1996-05-06-15:36:30;
  dtDate1   : LDATE_AND_TIME := LDT#1972-03-29-00:00:00;
  dtDate2   : LDT             := LDT#2018-08-08-13:33:20.5;

  dtEarliest : LDATE_AND_TIME := LDT#1979-1-1-00:00:00; // 0
  dtLatest   : LDATE_AND_TIME := LDT#2266-4-10-23:59:59; // 16#7FFF63888C620000
END_VAR
```

32-bit dates 'TIME_OF_DAY'

Use the TIME_OF_DAY (TOD) keyword to specify time.

Syntax:

<time keyword>#<time value>

<time keyword> : TIME_OF_DAY | time_of_day | TOD | tod
 <time value> : <hour>:<minute>:<second>
 <hour> : 0-23
 <minute> : 0-59
 <second> : 0.000-59.999

You can also specify fractions of a second for seconds. TIME_OF_DAY literals are internally treated as DWORD and thus the value is resolved in milliseconds.

Example:

```
PROGRAM MAIN
VAR
  tdClockTime0 : TIME_OF_DAY := TIME_OF_DAY#15:36:30.123;
  tdClockTime1 : TOD         := TOD#12:34:56.789;

  tdEarliest   : TIME_OF_DAY := TIME_OF_DAY#0:0:0.000;
  tdLatest     : TIME_OF_DAY := TIME_OF_DAY #23:59:59.999;
END_VAR
```

64-bit dates 'LTIME_OF_DAY

Use the keyword LTIME_OF_DAY (LTOD) to specify the time.

Syntax:

```
<time keyword>#<time value>
<time keyword> : LTIME_OF_DAY | ltime_of_day | LTOD | ltod
<time value>   : <hour>:<minute>:<second>
<hour>         : 0-23
<minute>       : 0-59
<second>       : 0.000-59.999999999
```

You can also specify fractions of a second for seconds. LTIME_OF_DAY literals are internally treated as LWORD, thus the value is resolved in nanoseconds.

Example:

```
PROGRAM MAIN
VAR
  tdClockTime0 : LTIME_OF_DAY := LTIME_OF_DAY#15:36:30.123;
  tdClockTime1 : LTOD         := LTOD#12:34:56.7890123456;

  tdEarliest   : LTIME_OF_DAY := LTIME_OF_DAY#0:0:0.000;
  tdLatest     : LTIME_OF_DAY := LTIME_OF_DAY#23:59:59.999999999;
END_VAR
```

See also:

- [Date and time data types \[► 761\]](#)

16.4.7 Typed Literals

With the exception of REAL/LREAL constants (where LREAL is always used), TwinCAT uses the smallest possible data type for calculations involving IEC constants. If you want to use a different data type, you can do this by using typed literals (typed constants), without having to declare the constant explicitly. Assign a prefix to the constant that specifies the type.

Syntax: <Type>#<Literal>

<Type> specifies the desired data type; possible entries are: BOOL, SINT, USINT, BYTE, INT, UINT, WORD, DINT, UDINT, DWORD, REAL, LREAL. You have to capitalize the type.

<Literal> specifies the constant. The input must match the data type specified under <Type>.

Example:

```
var1:=DINT#34;
```

If TwinCAT cannot transfer the constant to the target type without losing data, an error message is issued.

You can use typed constants wherever you can use normal constants.

16.4.8 Access to Variables in Arrays, Structures, and Blocks

Notation for access to:

- Two-dimensional array component: <Array-Name> [<1st dimension>, <2nd dimension>]

- **Structure variable:** <structure name> . <component name>
- **Function block or program variable:** <function block name> | <project name > . <variable name>

See also:

- [ARRAY \[► 773\]](#)
- [Structure \[► 778\]](#)
- [Object Function block \[► 84\]](#)
- [Object Program \[► 86\]](#)

16.4.9 Bit Access to Variables

With an index access you can address individual bits in integer variables. You can use a structure variable or a function block instance to address individual bits symbolically.

Index access to bits in integer variables

It is possible to address individual bits in integer variables. To do this, append the index of the bit to be addressed to the variable, separated by a dot. The bit index can be specified by any constant. Indexing is 0-based.

Syntax:

```
<integer variable name> . <index>
```

```
<integer data typ> = BYTE | WORD | DWORD | LWORD | SINT | USINT | INT | UINT | DINT | UDINT | LINT | ULINT
```

If the type of the variable is not allowed, TwinCAT issues the following error message: Invalid data type <type> for direct indexing. If the index is larger than the bit width of the variable, TwinCAT issues the following error: Index <n> out of valid range for variable <name>.

Sample index access:

In the program the third bit of the variable `nVarA` is set to the value of the variable `nVarB`.

```
PROGRAM MAIN
VAR
    nVarA : WORD := 16#FFFF;
bVarB : BOOL := 0;
END_VAR

// Index access in an integer variable
nVarA.2 := bVarB;
```

Result: `nVarA = 2#1111_1111_1111_1011 = 16#FFFB`

Sample constant as index:

The constant `cEnable` acts as an index to access the third bit of the variable `nVar`.

```
// GVL declaration
VAR_GLOBAL CONSTANT
    cEnable : USINT := 2;
END_VAR

PROGRAM MAIN
VAR
    nVar : INT := 0;
END_VAR

// Constant as index
nVar.cEnable := TRUE; // Third bit in nVar is set TRUE
```

Result: `nVar = 4`

● Accessibility of the variable defined by the bit number

i

Note that the variable defining the bit number (in the sample above `nEnable`) must be directly accessible via the variable name and without any preceding namespace.

So, for example, bit access is allowed if the variable is declared in the local scope (same level as `nVar`) or in the global scope on a GVL without the 'qualified_only' [▶ 822] attribute.

If the variable is declared on a GVL that has the 'qualified_only' attribute, access to `nEnable` is only possible by specifying the global variable list name (GVL.`nEnable`). Such an access to a global variable cannot be used for bit access (not possible: `nVar.GVL.nEnable := TRUE;`).

Symbolic bit access in structure variables

With the data type `BIT` you can designate individual bits with a name and combine them into a structure. The bit is then addressed with the component name.

Sample

Type declaration of the structure:

```
TYPE ST_ControllerData :
STRUCT
  nStatus_OperationEnabled : BIT;
  nStatus_SwitchOnActive   : BIT;
  nStatus_EnableOperation  : BIT;
  nStatus_Error            : BIT;
  nStatus_VoltageEnabled   : BIT;
  nStatus_QuickStop       : BIT;
  nStatus_SwitchOnLocked   : BIT;
  nStatus_Warning          : BIT;
END_STRUCT
END_TYPE
```

Declaration and write access to a bit:

```
PROGRAM MAIN
VAR
  stControllerDrive1 : ST_ControllerData;
END_VAR

// Symbolic bit access to nStatus_EnableOperation
stControllerDrive1.nStatus_EnableOperation := TRUE;
```

Symbolic bit access in function block instances

In function blocks you can declare variables for individual bits.

Sample

Type declaration of the structure:

```
FUNCTION_BLOCK FB_Controller
VAR_INPUT
  nSwitchOnActive   : BIT;
  nEnableOperation  : BIT;
  nVoltageEnabled   : BIT;
  nQuickStop       : BIT;
  nSwitchOnLocked   : BIT;
END_VAR
VAR_OUTPUT
  nOperationEnabled : BIT;
  nError            : BIT;
  nWarning          : BIT;
END_VAR
VAR
END_VAR
```

```
PROGRAM MAIN
VAR
  fbController : FB_Controller;
END_VAR
```

```
// Symbolic bit access to nSwitchOnActive
fbController(nSwitchOnActive:= TRUE);
```

See also:

- [BIT \[▶ 759\]](#)
- [Integer Data Types \[▶ 758\]](#)
- [Bit access in structures \[▶ 780\]](#)

Also see about this

- [Structure \[▶ 780\]](#)

16.4.10 Addresses

⚠ CAUTION

Shifting the contents of addresses by Online Change

If you use address pointers, address content may shift in the event of an Online Change!

● Automatic addressing

i It is recommended not to use direct addressing for allocated variables, but to use the * placeholder instead.

If the placeholders * (%I*, %Q* or %M*) are used, TwinCAT automatically performs flexible and optimized addressing.

Syntax:

```
<identifier> AT <address> : <data type>;
```

If an address is specified, the position in the memory and the size are expressed via special strings. An address is marked with the percent sign %, then follows the memory area prefix, the optional size prefix and the memory position.

```
%<memory area prefix> ( <size prefix> )? <memory position>
```

```
<memory area prefix> : I | Q | M
```

```
<size prefix> : X | B | W | D
```

```
<memory position> : * | <number> ( .<number> )*
```

Memory area prefix

I	Input memory area for inputs For physical inputs via input drivers ("sensors")
Q	Output memory area for outputs Physical outputs via output drivers ("actuators")
M	Flag memory area

Size prefix

X	Single bit
B	Byte (8 bits)
W	Word (16 bits)
D	Double word (32 bits)

Examples:

```
IbSensor1 AT%I* : BOOL;
IbSensor2 AT%IX7.5 : BOOL;
```

i If you do not explicitly specify a single bit address, Boolean variables are allocated byte by byte. Example: a value change of bVar AT %QB0 concerns the area from QX0.0 bis QX0.7.

Examples

Variable declarations:

lbSensor AT%I* : BOOL;	In the address specification, the placeholder * is specified instead of the memory position. This enables TwinCAT to perform flexible and optimized addressing automatically.
InInput AT%IW0 : WORD;	Variable declaration with address specification of an input word
ObActuator AT%QB0 : BOOL;	Boolean variable declaration Note: for Boolean variable one byte is allocated internally if no single bit address is specified. A value change of ObActuator consequently affects the range from QX0.0 to QX0.7.
lbSensor AT%IX7.5 : BOOL;	Boolean variable declaration with explicit specification of a single bit address. Only input bit 7.5 is read during access.

Other addresses:

%QX7.5	Single bit address of the output bit 7.5
%Q7.5	
%IW215	Word address of the input word 215
%QB7	Byte address of the output byte 7
%MD48	Address of a double word at memory location 48 in the flag area
%IW2.5.7.1	The interpretation depends on the current controller configuration (see below)

Possible memory area overlaps for direct addresses

In order to assign a valid address in a PLC project, you must know the desired position in the process image. For this, you first have to define the memory area and the required size. In selecting the memory position, the assignment of the different sizes in the memory as illustrated in the table below must be observed so that you can rule out memory area overlaps.

DWord	Word	Byte	Bit
D0	W0	B0	X0.0
		B1	X1.0
		B2	X2.0
		B3	X3.0
D1	W2	B4	X4.0
		B5	X5.0
		B6	X6.0
		B7	X7.0
D2	W4	B8	X8.0
		...	

Examples: memory area overlaps

1. W0 contains B0 and B1. If you place a Word variable at W0 and a Boolean variable at B1, the memory areas would overlap.
2. W3 contains B6 and B7. If you place a Word variable at W3 and a Boolean variable at B6, the memory areas would overlap.

See also:

- Programming a PLC Project > Declaring variables > [AT-Declaration](#) [► 69]

16.4.11 Functions

In ST, you can use a function call as an operand.

Example:

```
nResult := F_Add(7,5) + 3;
```

See also:

- [Object Function \[► 81\]](#)

TIME() Function

This function returns a value of data type TIME.

The TIME() function does not represent an absolute reference point, but it can be used for relative time measurements by calculating the difference between at least two TIME() return values.

Sample in ST:

The following example contains a TON block (fbTimer), for which a time of 5 seconds is applied (tTimerValue), and which is started at the start of the program. The rising edge of the timer activation is detected by an R_TRIG block (fbTrigger), whereupon the return value of the TIME() function is buffered for the first time (tTimeReturn1). When the time interval of the timer has elapsed, a second return value of the TIME() function is buffered (tTimeReturn2). The relative time between the respective TIME() calls is calculated via the between at least the two stored TIME() return values (tDifference). In this case the time between timer start and timer end is calculated as 5 seconds.

```
PROGRAM MAIN
VAR
  bStart          : BOOL := TRUE;
  fbTimer         : TON;
  tTimerValue     : TIME := T#5S;
  fbTrigger       : R_TRIG;
  tTimeReturn1   : TIME;
  tTimeReturn2   : TIME;
  tDifference     : TIME;
END_VAR

//=====

fbTimer(IN := bStart, PT := tTimerValue);
fbTrigger(CLK := fbTimer.IN);

IF fbTrigger.Q THEN
  tTimeReturn1 := TIME();
END_IF

IF fbTimer.Q THEN
  bStart      := FALSE;
  tTimeReturn2 := TIME();
  tDifference := tTimeReturn2 - tTimeReturn1; // The difference will be T#5s
END_IF
```

16.5 Data types

In programming, a variable is identified by its name and has an address in the target system's memory. Variable names are therefore identifiers under which the allocated storage space is addressed. The size of the variable is determined by its data type. This specifies how much storage space is reserved for the variable and how the values in memory are to be interpreted. The data type also determines which operations are allowed.

In TwinCAT there is also the possibility to instantiate function blocks. Function block instances then occupy memory in a similar way to variables. The memory requirement is determined by the function block.

The following groups of data types are available:

Standard data types

A standard data type is an elementary data type or a string data type.


```
<standard data type> : __UXINT | __XINT | __XWORD | BIT | BOOL | BYTE | DATE | DATE_AND_TIME | DINT
| DT | DWORD | INT | LDATE | LDATE_AND_TIME | LDT | LINT | LREAL | LTIME | LTOD | LWORD | REAL |
SINT | STRING | TIME | TOD | TIME_OF_DAY | UDINT | UINT | ULINT | USINT | WORD | WSTRING
```

See also:

- [BOOL \[▶ 758\]](#)
- [Integer Data Types \[▶ 758\]](#)
- [REAL/LREAL \[▶ 759\]](#)
- [STRING \[▶ 760\]](#)
- [WSTRING \[▶ 760\]](#)
- [TIME/LTIME \[▶ 760\]](#)
- [Date and time data types \[▶ 761\]](#)
- [Special data types XINT, UXINT, XWORD and PVOID \[▶ 767\]](#)

Extensions of the IEC 61131-3 standard

See also:

- [BIT \[▶ 759\]](#)
- [POINTER \[▶ 767\]](#)
- [REFERENCE \[▶ 770\]](#)
- [UNION \[▶ 784\]](#)
- [ANY and ANY <type> \[▶ 762\]](#)
- [Data type SYSTEM.ExceptionCode \[▶ 769\]](#)

User-defined data types

You can declare your own data types based on the default predefined ones or on existing data types.

Such data types are called user-defined or user-specific. The data types are either organized as a separate DUT object or declared within the declaration part of a programming object. They are also distinguished based on their purpose and syntax.

User-defined data type	Declaration	See also
Alias	DUT object	Alias [▶ 784]
Arrays	Programming object	ARRAY [▶ 773]
Enumeration	DUT object, programming object	Enumerations [▶ 781]
Pointer	Programming object	POINTER [▶ 767]
Reference	Programming object	REFERENCE [▶ 770]
Structure	DUT object	Structure [▶ 778]
Subrange type	Programming object	Subrange Types [▶ 758]
Union	DUT object	UNION [▶ 784]



Note the recommendations for naming identifiers.

See also:

- [Identifier \[▶ 844\]](#)

16.5.1 BOOL

Data type	Values	Memory space
BOOL	TRUE (1), FALSE (0)	8 bit

See also:

- [BOOL constants \[► 745\]](#)

16.5.2 Integer Data Types

The following integer data types are available in TwinCAT.

Data type	Lower bound	Upper bound	Memory space
BYTE	0	255	8 bit
WORD	0	65535	16 bit
DWORD	0	4294967295	32 bit
LWORD	0	$2^{64}-1$	64 bit
SINT	-128	127	8 bit
USINT	0	255	8 bit
INT	-32768	32767	16 bit
UINT	0	65535	16 bit
DINT	-2147483648	2147483647	32 bit
UDINT	0	4294967295	32 bit
LINT	-2^{63}	$2^{63}-1$	64 bit
ULINT	0	$2^{64}-1$	64 bit



When converting types from larger to smaller types, information may be lost.

See also:

- [Number constants \[► 745\]](#)

16.5.3 Subrange Types

A subrange type is a data type whose value range only covers a subset of a base type. Only integer types are possible as the base type.

Syntax: <Name> : <Inttype> (<ug>..<>og>)

<Name>	Valid IEC identifier
<Inttype>	Data type of the subrange (SINT, USINT, INT, UINT, DINT, UDINT, BYTE, WORD, DWORD, LINT, ULINT, LWORD).
<ug>	Lower bound of the range: Constant that must be compatible with the base data type. The lower bound itself is included in this range.
<og>	Upper bound of the range: Constant that must be compatible with the base data type. The upper bound itself is included in range.

Sample:

```
VAR
  nVarA: INT (-4095..4095);
  nVarB : UINT (0..10000);
END_VAR
```

If you assign a value to a subrange type in the declaration or implementation that does not fall within this range (for example `nVarA := 5000`), TwinCAT issues an error message.



Note the option to monitor the field bounds of a subrange type at runtime using the implicit monitoring functions `CheckRangeSigned` and `CheckRangeUnsigned`.

See also:

- [Range/LRange Checks \(POUs CheckRangeSigned, CheckRangeUnsigned, CheckLRangeSigned, CheckLRangeUnsigned\) \[► 167\]](#)
- [POU CheckRangeUnsigned](#)

16.5.4 BIT

You can only use the data type BIT for individual variables within structures or function blocks. The possible values are TRUE (1) and FALSE (0).

A BIT element requires 1 bit of memory space, and you can use it to address individual bits of a structure or function block using its name. BIT elements, which are declared sequentially, are consolidated to bytes. This allows you to optimize memory usage compared to BOOL types, which each occupy at least 8 bits. However, bit access takes significantly longer. Therefore, you should only use the data type BIT if you want to define the data in a specified format.

See also:

- [Structure \[► 778\]](#)
- [Object Function block \[► 84\]](#)

16.5.5 REAL/LREAL

The data types REAL and LREAL are floating-point types according to IEEE 754. They are necessary for the use of decimal numbers and floating-point numbers in point representation or exponential representation.

Data type	Lower limit	Upper limit	Smallest absolute value	Storage space
REAL	-3.402823e+38	3.402823e+38	1.0e-44	32-bit
LREAL	-1.7976931348623158e+308	1.7976931348623158e+308	4.94065645841247e-324	64-bit

Sample

```
PROGRAM MAIN
VAR
  fMaxReal      : REAL := 3.402823E+38; // Largest REAL number
  fPosMinReal   : REAL := 1.0E-44; // Smallest positive REAL number
  fNegMaxReal   : REAL := -1.0E-44; // Largest negative REAL number
  fMinReal      : REAL := -3.402823E+38; // Smallest REAL number

  fMaxLreal     : LREAL := 1.7976931348623157E+308; // Largest LREAL number
  fPosMinLreal  : LREAL := 4.94065645841247E-324; // Smallest positive LREAL number
  fNegMaxLreal  : LREAL := -4.94065645841247E-324; // Largest negative LREAL number
  fMinLreal     : LREAL := -1.7976931348623157E+308; // Smallest LREAL number
END_VAR
```



If the value of the REAL/LREAL number lies outside of the value range of the integer, an undefined result will be delivered when converting the data type from REAL or LREAL to SINT, USINT, INT, UINT, DINT, UDINT, LINT or ULINT.

i Allocation of a particularly large literal

In order to assign an integer literal that is larger than the upper limit of ULINT, either a comma must be set or an explicit typecast must be specified. Without such a specification as a floating-point number, information can be lost.

Example:

```
fMyReal : REAL := 3400000000000000000000000000000000.0;
fMyReal : REAL := REAL#3400000000000000000000000000000000;
```

See also:

- [REAL/LREAL constants \[► 746\]](#)

16.5.6 STRING

A variable of data type STRING can accept any string. The size specification for the storage space allocation in the declaration refers to characters and is enclosed by round or square brackets. If no size is specified, TwinCAT assumes 80 characters by default.

As a rule, TwinCAT does not limit the string length, but the string function only processes lengths between 1 and 255! If a variable is initialized with a string that is too long for the data type of the variable, TwinCAT truncates the string from the right.

i The storage space required for a STRING variable is always 1 byte per character + 1 additional byte, e.g. 81 bytes for a "STRING(80)" declaration.

Sample: string declaration with 35 characters

```
sVar : STRING(35) := 'This is a String';
```

See also:

- [STRING constants \[► 747\]](#)
- [WSTRING \[► 760\]](#)

16.5.7 WSTRING

Unlike the data type STRING (ASCII), the data type WSTRING is interpreted in Unicode format. This coding means that with WSTRING, the number of characters that can be displayed depends on which characters are to be displayed. With WSTRING, a length of 10 means that the length of the WSTRING can occupy a maximum of 10 WORDs. In Unicode, however, for some characters several WORDs are required for encoding a character, so that the number of characters does not have to correspond to the length of the WSTRING (10 in this case).

The data type requires 1 WORD per character and 1 WORD extra memory space. A STRING requires only 1 byte. The data type WSTRING is terminated with 0.

Example:

```
wsVar : WSTRING := "This is a WString";
```

See also:

- [STRING \[► 760\]](#)
- [STRING constants \[► 747\]](#)

16.5.8 TIME/LTIME

The time data type TIME is handled internally like a UDINT (32-bit). This leads to a resolution in milliseconds.

The time data type LTIME is internally handled like a ULINT (64-bit). You can use this data type as a time base for high-resolution timers with a resolution in nanoseconds.

Data type	Lower limit	Upper limit	Storage space	Resolution
TIME	0	4294967295 (49d17h2m47s295ms)	32-bit	Milliseconds
LTIME	0	213503d23h34m33s709ms551us615ns	64-bit	Nanoseconds

The time declaration can contain the time units that are valid for TIME or LTIME constants.

Sample:

```
VAR
  tTime : TIME := T#1d2h30m40s500ms
  tLTime : LTIME := LTIME# 100d2h30m40s500ms600us700ns
END_VAR
```

See also:

- [TIME/LTIME constants \[► 747\]](#)
- [Date and time data types \[► 761\]](#)
- [Date and time constants \[► 749\]](#)
- [TIME/TOD_TO <type> \[► 727\]](#)

16.5.9 Date and time data types

The DATE, DATE_AND_TIME (DT) and TIME_OF_DAY (TOD) data types are handled internally like a UDINT (32-bit value).

Data type	Lower limit	Upper limit	Storage space	Resolution
DATE	0 = D#1970-01-01 (01.01.1970)	4294967295 = D#2106-02-07 (07.02.2106)	32-bit	Seconds, although only the day is displayed.
DATE_AND_TIME DT	0 = DT#1970-1-1-0:0:0 (01.01.1970, 00:00 h)	4294967295 = DT#2106-02-07-06:28:15 (07.02.2106, 6:28:15)	32-bit	Seconds
TIME_OF_DAY TOD	0 = TOD#0:0:0 (00:00:00:000 h)	86399999 = TOD#23:59:59.999 (23:59:59.999 h)	32-bit	Milliseconds

The LDATE, LDATE_AND_TIME (LDT) and LTIME_OF_DAY (LTOD) are handled internally like a ULINT (64-bit).

Data type	Lower limit	Upper limit	Storage space	Resolution
LDATE	0 = LD#1970-1-1 (01.01.1970)	2 ⁶⁴ -1 = LD#2554-7-21 (21.07.2554)	64-bit	Nanoseconds, although only the day is displayed.
LDATE_AND_TIME LDT	0 = LDT#1970-1-1-0:0:0 (01.01.1970, 00:00 h)	2 ⁶⁴ -1 = LDT#2554-7-21-23:34:33.709551615 (21.07.2554, 23:34:33.709551615 h)	64-bit	Nanoseconds
LTIME_OF_DAY LTOD	0 = LTOD#0:0:0 (00:00:00:000 h)	86399999999999 = LTOD#23:59:59.999999999 (23:59:59.999999999 h)	64-bit	Nanoseconds

i Requirements

For the data types LDATE, LDATE_AND_TIME (LDT) and LTIME_OF_DAY (LTOD) TwinCAT version 3.1.4026.0 or later is required.

Samples:

```
VAR
// Date
dLowerLimit   : DATE           := DATE#1970-1-1;
dUpperLimit   : DATE           := DATE#2106-2-7;
dAppointment  : DATE           := D#2020-2-7;

// Date and time
dtLowerLimit  : DATE_AND_TIME := DATE_AND_TIME#1970-1-1-0:0:0;
dtUpperLimit  : DATE_AND_TIME := DATE_AND_TIME#2106-02-07-06:28:15;
dtAppointment : DT             := DT#2020-2-7-12:55:1.234;

// Time of day
tdLowerLimit  : TIME_OF_DAY    := TIME_OF_DAY#0:0:0;
tdUpperLimit  : TIME_OF_DAY    := TIME_OF_DAY#23:59:59.999;
tdAppointment : TOD            := TOD#12:3:4.567;

// Long date
dLowerLimit   : LDATE          := LDATE#1970-1-1;
dUpperLimit   : LDATE          := LDATE#2106-2-7;
dAppointment  : LDATE          := LD#2020-2-7;

// Long date and time
dtLowerLimit  : LDATE_AND_TIME := LDATE_AND_TIME#1970-1-1-0:0:0;
dtUpperLimit  : LDATE_AND_TIME := LDATE_AND_TIME#2262-4-10-23:59:59.99999999;
dtAppointment : LDT            := LDT#2020-2-7-12:55:1.234567891;

// Long time of day
tdLowerLimit  : LTIME_OF_DAY   := LTIME_OF_DAY#0:0:0;
tdUpperLimit  : LTIME_OF_DAY   := LTIME_OF_DAY#23:59:59.999999999;
tdAppointment : LTOD           := LTOD#12:3:4.567890123;

END_VAR
```

See also:

- [Date and time constants \[► 749\]](#)
- [TIME/LTIME \[► 760\]](#)
- [TIME/LTIME constants \[► 747\]](#)
- [DATE/DT TO <type> \[► 728\]](#)
- FileTime data type [T_FILETIME64](#) from the Tc2_Utilities PLC library
- DC Time data type [T_DCTIME64](#) from Tc2_EtherCAT PLC library

16.5.10 ANY and ANY_<type>

When implementing a function, a method or a function block (from build 4026), you can declare inputs (VAR_INPUT) as variables with generic IEC date type ANY or ANY_<type>. Consequently you can implement calls whose call parameters differ by their data type.

At runtime you can query the value transferred and its data type via a predefined structure within the programming block for the input variable.

The compiler internally replaces the type of the input variable with the data structure described below, but the value is not directly transferred. Instead, a pointer to the actual value is transferred, for which reason only one variable can be transferred. Therefore, the data type is only concretized at the call. Calls of such programming blocks can therefore take place with arguments that each have different data types.

An input of the data type ANY or ANY_<type> can neither be assigned any constants nor any property when calling the function, the function block or the method. Conversely, a property cannot be assigned any variable of the data type ANY or ANY_<type>.

The generic IEC data types shown below are supported. The table shows which generic data types allow which elementary data types.

Generic data types		Elementary data types				
ANY	ANY_BIT	<ul style="list-style-type: none"> • BYTE • WORD • DWORD • LWORD 				
	ANY_DATE	<ul style="list-style-type: none"> • DATE_AND_TIME, DT • DATE • TIME_OF_DAY, TOD • LDATE • LDATE_AND_TIME, LDT • LTIME_OF_DAY, LTOD 				
	ANY_NUM	<table border="0"> <tr> <td>ANY_REAL</td> <td> <ul style="list-style-type: none"> • REAL • LREAL </td> </tr> <tr> <td>ANY_INT</td> <td> <ul style="list-style-type: none"> • USINT • UINT • UDINT • ULINT • SINT • INT • DINT • LINT </td> </tr> </table>	ANY_REAL	<ul style="list-style-type: none"> • REAL • LREAL 	ANY_INT	<ul style="list-style-type: none"> • USINT • UINT • UDINT • ULINT • SINT • INT • DINT • LINT
	ANY_REAL	<ul style="list-style-type: none"> • REAL • LREAL 				
ANY_INT	<ul style="list-style-type: none"> • USINT • UINT • UDINT • ULINT • SINT • INT • DINT • LINT 					
ANY_STRING	<ul style="list-style-type: none"> • STRING • WSTRING 					

Internal data structure with 'ANY' and 'ANY_<type>'

When compiling the code, the input variables with ANY data type are replaced internally with the following structure. The structure elements are assigned to the actual call parameter at runtime.

```

TYPE AnyType :
STRUCT
  // the type of the actual parameter
  typeclass : __SYSTEM.TYPE_CLASS ;
  // the pointer to the actual parameter
  pvalue : POINTER TO BYTE;
  // the size of the data, to which the pointer points
  diSize : DINT;
END_STRUCT
END_TYPE
    
```



Via this structure you can access the input variable inside the programming block and, for example, query the transferred value.

Declaration

The syntax descriptions refer to a programming block with precisely one parameter (one input parameter).

Syntax

```

FUNCTION | FUNCTION_BLOCK | METHOD <POU name>
( : <return data type> )?
VAR_INPUT
  <input variable name> : <generic data type>;
END_VAR
<generic data type> = ANY | ANY_BIT | ANY_DATE |
ANY_NUM | ANY_REAL | ANY_INT | ANY_STRING
    
```

Call

The syntax descriptions refer to a programming block with precisely one parameter, to which an argument is transferred. The data type of the argument concretizes the generic data type of the input variable. For example, arguments of the type `BYTE`, `WORD`, `DWORD`, `LWORD` can be transferred to a `ANY_BIT` input variable.

Function call syntax

```
<variable name> := <function name> ( <argument name> );
<argument name> : variable with valid data type
```

Function block call syntax

```
<function block name> ( <input variable name> := <argument name> );
```

Method call syntax

```
<function block name> . <method name> ( <input variable name> := <argument name> );
```

Sample 1: transfer of elementary data types to inputs with generic data type

```
FUNCTION F_ComputeAny : BOOL
VAR_INPUT
    anyInput1      : ANY; // valid data type see table
END_VAR

FUNCTION_BLOCK FB_ComputeAny
VAR_INPUT
    anyInput1      : ANY;
END_VAR

FUNCTION_BLOCK FB_ComputeMethod
METHOD ComputeAny : BOOL
VAR_INPUT
    anyInput1      : ANY_INT; // valid data types are SINT, INT, DINT, USINT, UINT, UDINT, ULINT
END_VAR

PROGRAM PLC_PRG
VAR
    fbComputeAnyByte : FB_ComputeAny;
    fbComputeAnyInt  : FB_ComputeAny;

    fbComputeM1      : FB_ComputeMethod;
    fbComputeM2      : FB_ComputeMethod;

    nByte             : BYTE := 16#AB;
    nInt               : INT  := -1234;
    bResultByte       : BOOL;
    bResultInt         : BOOL;
END_VAR

bResultByte := F_ComputeAny(nByte);
bResultInt  := F_ComputeAny(nInt);

fbComputeAnyByte(anyInput1 := nByte);
fbComputeAnyInt(anyInput1 := nInt);

fbComputeM1.methComputeAny(anyInput1 := nByte);
fbComputeM2.methComputeAny(anyInput1 := nInt);
```

Sample 2: transfer of elementary data types to inputs with generic data type

The transfer parameters of the function calls have different data types.

```
FUNCTION F_AnyBitFunc      : BOOL
VAR_INPUT
    value : ANY_BIT;
END_VAR

FUNCTION F_AnyDateFunc    : BOOL
VAR_INPUT
    value : ANY_DATE;
END_VAR

FUNCTION F_AnyFunc        : BOOL
VAR_INPUT
    value : ANY;
END_VAR

FUNCTION F_AnyIntFunc     : BOOL
```



```

VAR_INPUT
    value : ANY_INT;
END_VAR

FUNCTION F_AnyNumFunc      : BOOL
VAR_INPUT
    value : ANY_NUM;
END_VAR

FUNCTION F_AnyRealFunc    : BOOL
VAR_INPUT
    value : ANY_REAL;
END_VAR

FUNCTION F_AnyStringFunc  : BOOL
VAR_INPUT
    value : ANY_STRING;
END_VAR

PROGRAM MAIN
VAR
    bBOOL      : BOOL      := TRUE;
    nBYTE      : BYTE      := 16#AB;
    nWORD      : WORD      := 16#1234;
    nDWORD     : DWORD     := 16#6789ABCD;
    nLWORD     : LWORD     := 16#0123456789ABCDEF;
    sSTRING    : STRING    := 'xyz';
    wsWSTRING  : WSTRING   := "abc";
    dtDATEANDTIME : DATE_AND_TIME := DT#2017-02-20-11:07:00;
    dDATE      : DATE      := D#2017-02-20;
    tdTIMEOFDAY : TIME_OF_DAY := TOD#11:07:00;
    fREAL      : REAL      := 42.24;
    flREAL     : LREAL     := 24.42;
    nUSINT     : USINT     := 12;
    nUINT      : UINT      := 1234;
    nUDINT     : UDINT     := 12345;
    nULINT     : ULINT     := 123456;
    nSINT      : SINT      := -12;
    nINT       : INT       := -1234;
    nDINT      : DINT      := -12345;
    nLINT      : LINT      := -123456;
END_VAR

F_AnyFunc (bBOOL);
F_AnyFunc (nBYTE);
F_AnyFunc (nWORD);
F_AnyFunc (nDWORD);
F_AnyFunc (nLWORD);
F_AnyFunc (sSTRING);
F_AnyFunc (wsWSTRING);
F_AnyFunc (dtDATEANDTIME);
F_AnyFunc (tdTIMEOFDAY);
F_AnyFunc (fREAL);
F_AnyFunc (flREAL);
F_AnyFunc (nUSINT);
F_AnyFunc (nUINT);
F_AnyFunc (nUDINT);
F_AnyFunc (nULINT);
F_AnyFunc (nSINT);
F_AnyFunc (nINT);
F_AnyFunc (nDINT);
F_AnyFunc (nLINT);

F_AnyBitFunc (nBYTE);
F_AnyBitFunc (nWORD);
F_AnyBitFunc (nDWORD);
F_AnyBitFunc (nLWORD);

F_AnyStringFunc (sSTRING);
F_AnyStringFunc (wsWSTRING);

F_AnyDateFunc (dtDATEANDTIME);
F_AnyDateFunc (dDATE);
F_AnyDateFunc (tdTIMEOFDAY);

F_AnyNumFunc (fREAL);
F_AnyNumFunc (flREAL);
F_AnyNumFunc (nUSINT);
F_AnyNumFunc (nUINT);
F_AnyNumFunc (nUDINT);
F_AnyNumFunc (nULINT);

```

```

F_AnyNumFunc (nSINT);
F_AnyNumFunc (nINT);
F_AnyNumFunc (nDINT);
F_AnyNumFunc (nLINT);

F_AnyRealFunc (fREAL);
F_AnyRealFunc (fLREAL);

F_AnyIntFunc (nUSINT);
F_AnyIntFunc (nUINT);
F_AnyIntFunc (nUDINT);
F_AnyIntFunc (nULINT);
F_AnyIntFunc (nSINT);
F_AnyIntFunc (nINT);
F_AnyIntFunc (nDINT);
F_AnyIntFunc (nLINT);

```

Sample 3: comparison of two transferred variables

The function compares the two variables transferred to determine whether they are of the same type and have the same value.

```

FUNCTION F_GenericCompare : BOOL
VAR_INPUT
    any1 : ANY;
    any2 : ANY;
END_VAR
VAR
    nCount: DINT;
END_VAR

IF any1.typeclass <> any2.typeclass THEN
    RETURN;
END_IF

IF any1.diSize <> any2.diSize THEN
    RETURN;
END_IF

// Byte comparison
FOR nCount := 0 TO any1.diSize-1 DO
    IF any1.pvalue[nCount] <> any2.pvalue[nCount] THEN
        RETURN;
    END_IF
END_FOR

F_GenericCompare := TRUE;

```

Sample 4: determination of the transferred data type

The function checks whether the transferred variable is of the type REAL or LREAL. If this is the case, the value of the variable is rounded.

```

// function to round transfer parameters of the type REAL and LREAL (other types are detected as
invalid)
FUNCTION F_RoundFloatingValue : INT
VAR_INPUT
    anyIn      : ANY;           // input variable of the type ANY
END_VAR
VAR
    pAnyReal   : POINTER TO REAL; // pointer to a variable of the type REAL
    pAnyLReal  : POINTER TO LREAL; // pointer to a variable of the type LREAL
END_VAR
VAR_OUTPUT
    bInvalidType : BOOL;           // output variable with value TRUE if the transferred
parameter has an invalid type
END_VAR

// round floating value for a transfer parameter of the type REAL
IF (anyIn.TypeClass = __SYSTEM.TYPE_CLASS.TYPE_REAL) THEN
    pAnyReal      := anyIn.pValue;
    F_RoundFloatingValue := REAL_TO_INT(pAnyReal^);

// round floating value for a transfer parameter of the type LREAL
ELSIF (anyIn.TypeClass = __SYSTEM.TYPE_CLASS.TYPE_LREAL) THEN
    pAnyLReal     := anyIn.pValue;
    F_RoundFloatingValue := LREAL_TO_INT(pAnyLReal^);

// inform about invalid type if the transfer parameter is not of the type REAL or LREAL

```

```
ELSE
    bInvalidType      := TRUE;
END_IF
```

16.5.11 Special data types XINT, UXINT, XWORD and PVOID

Variables with these data types are converted to a platform-compliant data type, depending on the target system.

TwinCAT supports systems with address register widths between 32 and 64 bits. In order to make the IEC code as independent as possible from the target system, you can use the "pseudo" data types `UXINT`, `XINT`, `XWORD` and `PVOID` listed below. The compiler checks which target system type is currently used and converts these data types to the respective standard data types. In addition, type conversion operators are available for variables of this data type.

The following "pseudo" data types are available:

	Type conversion on 64-bit platforms	Type conversion on 32-bit platforms
XINT or <code>__XINT</code>	LINT	DINT
UXINT or <code>__UXINT</code>	ULINT	UDINT
XWORD or <code>__XWORD</code>	LWORD	DWORD
PVOID	UXINT	

16.5.12 POINTER

At runtime, a pointer saves the memory address of objects such as variables or function block instances.

Syntax

```
<pointer name>: POINTER TO <data type> | <data unit type> | <function block name>;
```

Sample

```
FUNCTION_BLOCK FB_Sample
VAR
    pSample : POINTER TO INT;
    nVar1 : INT := 5;
    nVar2 : INT;
END_VAR

pSample := ADR(nVar1); // pointer pSample is assigned to address of nVar1
nVar2 := pSample^;    //
// value 5 of nVar1 is assigned to variable nVar2 by dereferencing of pointer pSample
```

Dereferencing a pointer means obtaining the value to which the pointer points. A pointer is dereferenced by appending the content operator `^` to the pointer identifier, for example `pSample^` in the sample shown above. To assign the address of an object to a pointer, use the address operator `ADR`: `ADR(nVar1)`.

In online mode, you can use the [Go to reference \[► 880\]](#) command to jump from a pointer to the declaration location of the referenced object (available from TC3.1 Build 4026).



If a pointer to an allocated input variable is used, the access is interpreted as write access. This leads to the compiler warning "<Pointer Name> is not a valid assignment target" during code generation.

Sample: `pTest := ADR(nInput);`

If you need a construct of this type, you must first copy the input value (`nInput`) to a variable with write access.

Index access to pointers

TwinCAT allows index access [] to variables of type POINTER, as well as to the data types STRING or WSTRING.

The data to which the pointer points can also be accessed by appending the bracket operator [] to the pointer identifier, for example `pData[i]`. The basic data type of the pointer determines the data type and the size of the indexed component. The index access to the pointer takes place arithmetically by adding the index-dependent offset $i * \text{sizeof}(\langle \text{base type} \rangle)$ to the address of the pointer. The pointer is implicitly dereferenced at the same time. Calculation: `pData[i] := (pData + i * sizeof(INT))^;`

Index access to STRING

If you use index access for a variable of type STRING, you get the character at the offset of the index expression. The result is of type BYTE. For example, `sData[i]` returns the *i*th character of the string `sData` as SINT (ASCII).

Index access WSTRING

If you use index access for a variable of type WSTRING, you get the character at the offset of the index expression. The result is of type WORD. For example, `wsData[i]` returns the *i*th character of the string as INT (Unicode).

Subtracting pointers

The result of the difference between two pointers is a value of the type DWORD, even on 64-bit platforms, if the pointers are 64-bit pointers.



Note the possibility of using references. The advantage of using references is that type safety is guaranteed. That is not the case with pointers.



The memory access of pointers during runtime can be checked by the implicit monitoring function `CheckPointer`.

Automatic pointer/reference update during Online Change



Available from TwinCAT 3.1 Build 4026

The following descriptions refer to both pointers and references. For ease of reading, however, only the term pointer is used in the following.

Function:

During an Online Change, the values of all PLC pointers are automatically updated so that the respective pointer refers to the same variable or the same object as before the Online Change. This means that a pointer remains valid after the Online Change, even if the variable it points to is moved to a different memory position during the Online Change.

Mode of operation:

During an Online Change, a check is made for each pointer to see whether it refers to a PLC symbol. The following requirements must be met for such a determination.

- Case 1: Symbol was determined
 - If the symbol still exists in the new symbols that exist after the Online Change and if the type of the symbol has not changed, the pointer is moved to the symbol found. The pointer or address value is updated.
 - If the symbol no longer exists in the new symbols or if the type of symbol has changed, the pointer is set to zero.

- Case 2: Symbol was not determined
 - If a pointer refers to an address for which no symbol is found before the Online Change (e.g. because the pointer points to a memory area outside the PLC), the pointer value is not changed during the Online Change.

Requirements:

- This function requires the (ADS) symbol description of the variable to which the pointer points to. If there is no symbol description for a variable, e.g. due to the use of Attribute 'hide' [▶ 805], the pointer is not taken into account.
- The pointer must point to a variable/object within the PLC memory. Pointers pointing to a memory area outside the PLC memory are not changed during Online Change.

Additional update option:

The functionality described is available from TwinCAT 3.1 Build 4026. Any previous, manually implemented mechanism for updating a pointer can still be used. The desired destination can be assigned to the pointer cyclically, for example. There is no need to remove these lines of code from the project. Similarly, it is not necessary to keep or implement these lines of code.

See also:

- REFERENCE [▶ 770]
- Pointer Checks (POU CheckPointer) [▶ 168]

16.5.13 Data type __SYSTEM.ExceptionCode

See also: __TRY, __CATCH, __FINALLY, __ENDTRY [▶ 737]

```

TYPE ExceptionCode :
(
  RTSEXCPT_UNKNOWN                := 16#FFFFFFF,
  RTSEXCPT_NOEXCEPTION            := 16#00000000,
  RTSEXCPT_WATCHDOG               := 16#00000010,
  RTSEXCPT_HARDWAREWATCHDOG       := 16#00000011,
  RTSEXCPT_IO_CONFIG_ERROR        := 16#00000012,
  RTSEXCPT_PROGRAMCHECKSUM        := 16#00000013,
  RTSEXCPT_FIELDBUS_ERROR         := 16#00000014,
  RTSEXCPT_IOUPDATE_ERROR         := 16#00000015,
  RTSEXCPT_CYCLE_TIME_EXCEED      := 16#00000016,
  RTSEXCPT_ONLCHANGE_PROGRAM_EXCEEDED := 16#00000017,
  RTSEXCPT_UNRESOLVED_EXTREFS     := 16#00000018,
  RTSEXCPT_DOWNLOAD_REJECTED      := 16#00000019,
  RTSEXCPT_BOOTPROJECT_REJECTED_DUE_RETAIN_ERROR := 16#0000001A,
  RTSEXCPT_LOADBOOTPROJECT_FAILED := 16#0000001B,
  RTSEXCPT_OUT_OF_MEMORY          := 16#0000001C,
  RTSEXCPT_RETAIN_MEMORY_ERROR    := 16#0000001D,
  RTSEXCPT_BOOTPROJECT_CRASH      := 16#0000001E,
  RTSEXCPT_BOOTPROJECTTARGETMISMATCH := 16#00000021,
  RTSEXCPT_SCHEDULEERROR          := 16#00000022,
  RTSEXCPT_FILE_CHECKSUM_ERR      := 16#00000023,
  RTSEXCPT_RETAIN_IDENTITY_MISMATCH := 16#00000024,
  RTSEXCPT_IEC_TASK_CONFIG_ERROR  := 16#00000025,
  RTSEXCPT_APP_TARGET_MISMATCH    := 16#00000026,
  RTSEXCPT_ILLEGAL_INSTRUCTION    := 16#00000050,
  RTSEXCPT_ACCESS_VIOLATION       := 16#00000051,
  RTSEXCPT_PRIV_INSTRUCTION       := 16#00000052,
  RTSEXCPT_IN_PAGE_ERROR          := 16#00000053,
  RTSEXCPT_STACK_OVERFLOW         := 16#00000054,
  RTSEXCPT_INVALID_DISPOSITION    := 16#00000055,
  RTSEXCPT_INVALID_HANDLE        := 16#00000056,
  RTSEXCPT_GUARD_PAGE             := 16#00000057,
  RTSEXCPT_DOUBLE_FAULT          := 16#00000058,
  RTSEXCPT_INVALID_OPCODE        := 16#00000059,
  RTSEXCPT_MISALIGNMENT           := 16#00000100,
  RTSEXCPT_ARRAYBOUNDS           := 16#00000101,
  RTSEXCPT_DIVIDEBYZERO          := 16#00000102,
  RTSEXCPT_OVERFLOW              := 16#00000103,
  RTSEXCPT_NONCONTINUABLE        := 16#00000104,
  RTSEXCPT_PROCESSORLOAD_WATCHDOG := 16#00000105,

```

```

RTSEXCPT_FPU_ERROR                := 16#00000150,
RTSEXCPT_FPU_DENORMAL_OPERAND    := 16#00000151,
RTSEXCPT_FPU_DIVIDEBYZERO        := 16#00000152,
RTSEXCPT_FPU_INEXACT_RESULT      := 16#00000153,
RTSEXCPT_FPU_INVALID_OPERATION   := 16#00000154,
RTSEXCPT_FPU_OVERFLOW            := 16#00000155,
RTSEXCPT_FPU_STACK_CHECK        := 16#00000156,
RTSEXCPT_FPU_UNDERFLOW          := 16#00000157,
RTSEXCPT_VENDOR_EXCEPTION_BASE  := 16#00002000,
RTSEXCPT_USER_EXCEPTION_BASE    := 16#00010000
) UDINT ;
END_TYPE

```

16.5.14 Interface pointer / INTERFACE

An interface pointer (also called an interface variable) stores the address of a function table or a function block instance that implements the interface. All methods and properties that are defined by the interface can be called via this indirection with an interface pointer. These methods and properties must therefore be implemented by the function block whose instance is assigned to the interface pointer.

An instance of a function block that implements a corresponding interface must always be assigned to an interface pointer before use.

It is recommended to check the interface pointer for validity ($\neq 0$) before use.

Syntax:

```

<Kennzeichner> : <Interfacetyp>;
FUNCTION_BLOCK <Funktionsbausteintyp> IMPLEMENTS <Interfacetyp>

```

Example:

```

FUNCTION_BLOCK FB_Sample IMPLEMENTS I_Sample
VAR
    ipSample : I_Sample;
    fbSample : FB_Sample;
    nResult  : INT;
END_VAR
ipSample := fbSample;

// calling a method of this interface
IF ipSample <> 0 THEN
    nResult := ipSample.Add(3, 6); // result is 9
END_IF

```

The use of interfaces and interface pointers is explained in detail in the section Object-oriented programming > [Object Interface](#) [▶ 102].

Type conversions are possible with the operator `__QUERYINTERFACE()` [▶ 735].

16.5.15 REFERENCE

A reference points implicitly to another object. The reference is implicitly dereferenced during the access and therefore requires no special content operator `^` like a pointer.

Syntax

```

<identifier> : REFERENCE TO <data type> ;
<data type> : base type of the reference

```

Sample declaration:

```

PROGRAM_MAIN
VAR
    refInt : REFERENCE TO INT;
    nA     : INT;
    nB     : INT;
END_VAR

```

You can now use `refInt` as "alias" for variables of type `INT`.

Assignment:

You have to set the address of the reference via a separate assignment operation with the help of the [assignment operator REF= \[► 771\]](#). An exception to this is when an input is a REFERENCE TO and the input is transferred within the call. In this case the normal allocation operator := is used instead of the allocation operator REF=.

```
FUNCTION_BLOCK FB_Sample
VAR_INPUT
    refInput1 : REFERENCE TO INT;
    refInput2 : REFERENCE TO INT;
END_VAR

PROGRAM MAIN
VAR
    fbSample : FB_Sample;
    n1       : INT;
    n2       : INT;
END_VAR
```

```
fbSample.refInput1 REF= n1;
fbSample(refInput2 := n2);
```

You can check whether a reference points to a valid value (e.g. not equal to 0) with the help of a special operator (see [Checking references for validity \[► 772\]](#)).

Application example:

```
refInt REF= nA; // refInt points now to nA
refInt := 12; // nA has got the value 12
nB := refInt * 2; // nB has got the value 24
refInt REF= nB; // refInt points now to nB
refInt := nA / 2; // nB has got the value 6
refInt REF= 0; // explicit initialisation of the reference
```



TwinCAT initializes references (with 0).



If a reference points to an allocated input variable, the access (e.g. rInput REF=Input;) is regarded as a write access. This is not possible and leads to a compiler error during code generation.

Allocation operator REF=

The operator generates a [reference \[► 770\]](#) (pointer) to a value.

Syntax:

```
<variable name> REF= <variable name>
```

Sample:

```
PROGRAM MAIN
VAR
    refA : REFERENCE TO ST_Sample;
    stA  : ST_Sample;
    refB : REFERENCE TO ST_Sample;
    stB1 : ST_Sample;
    stB2 : ST_Sample;
END_VAR

refA REF= stA; // represents => refA := ADR(stA);
refB REF= stB1; // represents => refB := ADR(stB1);
refA := refB; // represents => refA^ := refB^; (value assignment of refB as refA and refB are implicitly dereferenced)
refB := stB2; // represents => refB^ := stB2; (value assignment of stB2 as refB is implicitly dereferenced)
END_VAR
```

Invalid declarations

```
PROGRAM MAIN
VAR
    aTest : ARRAY[0..9] OF REFERENCE TO INT;
```

```
pTest      : POINTER TO REFERENCE TO INT;
refTestRef : REFERENCE TO REFERENCE TO INT;
refTestBit : REFERENCE TO BIT;
END_VAR
```

A reference type may not be used as the base type of an array, pointer or a reference. In addition, a reference may not point to a bit variable. Such constructs generate compiler errors.

Comparison of reference and pointer

In comparison with a pointer, a reference has the following advantages:

- Easier to use:
A reference can directly access the contents of the referenced object without dereferencing.
- Nicer and simpler syntax when transferring values:
Calling a function block that transfers a reference without address operator instead of a pointer.
Sample: `FB_Test1(refInput := nValue);`
instead of: `FB_Test2(pInput := ADR(nValue));`
- Type safety:
When assigning two references, the compiler checks whether the base types match. This is not checked in the case of pointers.

Checking references for validity

You can use the operator `__ISVALIDREF` to check whether a reference points to a valid value, i.e. a value not equal to 0.

Syntax:

```
<boolesche Variable> := __ISVALIDREF(<mit REFERENCE TO <datatype> deklarierter Kennzeichner>);
```

The boolean variable becomes TRUE if the reference points to a valid value, otherwise FALSE.

Sample

```
PROGRAM_MAIN
VAR
  nVar      : INT;
  refInt1   : REFERENCE TO INT;
  refInt2   : REFERENCE TO INT;
  bTestRef1 : BOOL := FALSE;
  bTestRef2 : BOOL := FALSE;
END_VAR

nVar      := nVar + 1;
refInt1 REF= nVar;
refInt2 REF= 0;
bTestRef1 := __ISVALIDREF(refInt1); (* becomes TRUE, because refInt1 points to nVar, which is non-zero *)
bTestRef2 := __ISVALIDREF(refInt2); (* becomes FALSE, because refInt2 is set to 0 *)
```



The implicit monitoring function Checkpointer acts on variables of the type REFERENCE in the same way as on pointer variables.

Automatic pointer/reference update during Online Change



Available from TwinCAT 3.1 Build 4026

The following descriptions refer to both pointers and references. For ease of reading, however, only the term pointer is used in the following.

Function:

During an Online Change, the values of all PLC pointers are automatically updated so that the respective pointer refers to the same variable or the same object as before the Online Change. This means that a pointer remains valid after the Online Change, even if the variable it points to is moved to a different memory position during the Online Change.

Mode of operation:

During an Online Change, a check is made for each pointer to see whether it refers to a PLC symbol. The following requirements must be met for such a determination.

- Case 1: Symbol was determined
 - If the symbol still exists in the new symbols that exist after the Online Change and if the type of the symbol has not changed, the pointer is moved to the symbol found. The pointer or address value is updated.
 - If the symbol no longer exists in the new symbols or if the type of symbol has changed, the pointer is set to zero.
- Case 2: Symbol was not determined
 - If a pointer refers to an address for which no symbol is found before the Online Change (e.g. because the pointer points to a memory area outside the PLC), the pointer value is not changed during the Online Change.

Requirements:

- This function requires the (ADS) symbol description of the variable to which the pointer points to. If there is no symbol description for a variable, e.g. due to the use of [Attribute 'hide' \[► 805\]](#), the pointer is not taken into account.
- The pointer must point to a variable/object within the PLC memory. Pointers pointing to a memory area outside the PLC memory are not changed during Online Change.

Additional update option:

The functionality described is available from TwinCAT 3.1 Build 4026. Any previous, manually implemented mechanism for updating a pointer can still be used. The desired destination can be assigned to the pointer cyclically, for example. There is no need to remove these lines of code from the project. Similarly, it is not necessary to keep or implement these lines of code.

See also:

- [Pointer Checks \(POU CheckPointer\) \[► 168\]](#)

Also see about this

 [POINTER \[► 767\]](#)

16.5.16 ARRAY

An array is a collection of data elements of the same data type. TwinCAT supports one-dimensional and multi-dimensional arrays of fixed or variable length.

16.5.16.1 Array with fixed length

Arrays can be declared in the declaration part of a POU or in global variable lists.

Syntax for declaring a one-dimensional array

```
<variable name> : ARRAY[ <dimension> ] OF <data type> ( := <initialization> )? ;
<dimension> : <lower index bound>..<upper index bound>
<data type> : elementary data types | user defined data types | function block types
// (...)?: Optional
```

Syntax for declaring a multidimensional array

```
<variable name> : ARRAY[ <1st dimension> ( , <next dimension> )
+ ] OF <data type> ( := <initialization> )? ;
```

```

<1st dimension> : <1st lower index bound>..<1st upper index bound>
<next dimension> : <next lower index bound>..<next upper index bound>
<data type> : elementary data types | user defined data types | function block types
// (...) + : One or more further dimensions
// (...) ? : Optional

```

The index limits are integer numbers up to data type DINT.

Syntax for data access

```

<variable name>[ <index of 1st dimension> ( , <index of next dimension> ) * ]
// (...) * : 0, one or more further dimensions

```



Note the option to monitor violation of field bounds at runtime using the implicit monitoring function CheckBounds.

See also:

- [Bound Checks \(POU CheckBounds\) \[► 164\]](#)



Setting the monitoring range for large arrays

If an array has more than 1000 elements, then 1000 elements are displayed by default in the online view. You can change this monitoring range by double-clicking the array declaration in the declaration section in the online view. A dialog then opens in which you can set the start and end index of the desired monitoring range.

Please note that the display performance is reduced, the more elements are simultaneously displayed (maximum 20000 elements).

Example: One-dimensional array

Declaration 1:

One-dimensional array of 10 integer elements

Lower index limit: 0

Upper index limit: 9

```

VAR
  aCounter : ARRAY[0..9] OF INT;
END_VAR

```

Initialization 1:

```

aCounter : ARRAY[0..9] OF INT := [0, 10, 20, 30, 40, 50, 60, 70, 80, 90];

```

Data access 1:

The local variable is assigned the value 20.

```

nLocalVariable := aCounter[2];

```

Declaration 2:

```

VAR CONSTANT
  cMin   : INT := 0;
  cMax   : INT := 5;
END_VAR
VAR
  aSample : ARRAY [cMin..cMax] OF BOOL;
END_VAR

```

Data access 2:

The array is accessed by means of an index variable. Before the array is accessed, the system checks whether the value of the index variable is within the valid array boundaries.

```

IF nIndex >= cMin AND nIndex <= cMax THEN
  bValue := aSample[nIndex];
END_IF

```

Example: Two-dimensional array**Declaration:**

First dimension: 1 to 2

Second dimension: 3 to 4

```
VAR
  aCardGame : ARRAY[1..2, 3..4] OF INT;
END_VAR
```

Initialization:

```
aCardGame : ARRAY[1..2, 3..4] OF INT := [2(10),2(20)]; // Short notation for [10, 10, 20, 20]
```

Data access:

```
nLocal1 := aCardGame[1, 3]; // Assignment of 10
nLocal2 := aCardGame[2, 4]; // Assignment of 20
```

Example: Three-dimensional array**Declaration:**

First dimension: 1 to 2

Second dimension: 3 to 4

Third dimension: 5 to 6

 $2 * 2 * 2 = 8$ array elements

```
VAR
  aCardGame : ARRAY[1..2, 3..4, 5..6] OF INT;
END_VAR
```

Initialization 1:

```
aCardGame : ARRAY[1..2, 3..4, 5..6] OF INT := [10, 20, 30, 40, 50, 60, 70, 80];
```

Data access 1:

```
nLocal1 := aCardGame[1, 3, 5]; // Assignment of 10
nLocal2 := aCardGame[2, 3, 5]; // Assignment of 20
nLocal3 := aCardGame[1, 4, 5]; // Assignment of 30
nLocal4 := aCardGame[2, 4, 5]; // Assignment of 40
nLocal5 := aCardGame[1, 3, 6]; // Assignment of 50
nLocal6 := aCardGame[2, 3, 6]; // Assignment of 60
nLocal7 := aCardGame[1, 4, 6]; // Assignment of 70
nLocal8 := aCardGame[2, 4, 6]; // Assignment of 80
```

Initialization 2:

```
aCardGame : ARRAY[1..2, 3..4, 5..6] OF INT := [2(10), 2(20), 2(30), 2(40)]; // Short notation for [10, 10, 20, 20, 30, 30, 40, 40]
```

Data access 2:

```
nLocal1 := aCardGame[1, 3, 5]; // Assignment of 10
nLocal2 := aCardGame[2, 3, 5]; // Assignment of 10
nLocal3 := aCardGame[1, 4, 5]; // Assignment of 20
nLocal4 := aCardGame[2, 4, 5]; // Assignment of 20
nLocal5 := aCardGame[1, 3, 6]; // Assignment of 30
nLocal6 := aCardGame[2, 3, 6]; // Assignment of 30
nLocal7 := aCardGame[1, 4, 6]; // Assignment of 40
nLocal8 := aCardGame[2, 4, 6]; // Assignment of 40
```

Example: Three-dimensional array of a user-defined structure**Declaration:**The array aData consists of a total of $3 * 3 * 10 = 90$ array elements of data type ST_Data.

```
TYPE ST_Data
STRUCT
  n1 : INT;
  n2 : INT;
  n3 : DWORD;
END_STRUCT
END_TYPE
```

```
PROGRAM MAIN
VAR
  aData : ARRAY[1..3, 1..3, 1..10] OF ST_Data;
END_VAR
```

Partial initialization:

In the example only the first 3 elements are explicitly initialized. Elements to which no initialization value is explicitly assigned are internally initialized with the default value of the basic data type. Thus the structure components are initialized with 0 starting from the element aData[2, 1, 1].

```
aData : ARRAY[1..3, 1..3, 1..10] OF ST_Data := [(n1 := 1, n2 := 10, n3 := 16#00FF),
                                                (n1 := 2, n2 := 20, n3 := 16#FF00),
                                                (n1 := 3, n2 := 30, n3 := 16#FFFF)];
```

Data access:

```
nLocal1 := aData[1,1,1].n1; // Assignment of 1
nLocal2 := aData[3,1,1].n3; // Assignment of 16#FFFF
```

Example: Array of a function block**Declaration:**

The array aObjects consists of 4 elements. Each element instantiates a function block FB_Object.

```
FUNCTION_BLOCK FB_Object
VAR
  nCounter : INT;
END_VAR

PROGRAM MAIN
VAR
  aObjects : ARRAY[1..4] OF FB_Object;
END_VAR
```

Function call:

```
aObjects[2]();
```

Example: Array of a function block with FB_init method**Declaration:**

Implementation of FB_Sample with method FB_init

```
FUNCTION_BLOCK FB_Sample
VAR
  _nId : INT;
  _fIn : LREAL;
END_VAR
```

The function block FB_Sample has a method FB_init that requires two parameters.

```
METHOD FB_init : BOOL
VAR_INPUT
  bInitRetains : BOOL;
  bInCopyCode : BOOL;
  nId           : INT;
  fIn          : LREAL;
END_VAR

_nId := nId;
_fIn := fIn;
```

Array declaration with initialization:

```
PROGRAM MAIN
VAR
  fbSample : FB_Sample (nId := 11, fIn := 33.44);
  aSample  : ARRAY[0..1, 0..1] OF FB_Sample [(nId := 12, fIn := 11.22),
                                              (nId := 13, fIn := 22.33),
                                              (nId := 14, fIn := 33.55),
                                              (nId := 15, fIn := 11.22)];
END_VAR
```

16.5.16.2 Array of arrays

The declaration of an "array of arrays" is an alternative notation for multidimensional arrays. Instead of dimensioning the elements, a collection of elements is nested. Any nesting depth can be used.

Syntax for declaration

```
<variable name> : ARRAY[<first>] ( OF ARRAY[<next>] )+ OF <data type> ( := <initialization> )? ;
<first> : <first lower index bound>..<first upper index bound>
<next> : <lower index bound>..<upper index bound> // one or more arrays
<data type> : elementary data types | user defined data types | function block types
// (...) + : One or more further arrays
// (...) ? : Optional
```

Syntax for data access

```
<variable name>[<index of first array>] ( [<index of next array>] )+ ;
// (...) * : 0, one or more further arrays
```

Example

The variables aiPoints and ai2Boxes combine the same data elements, but the spellings for declaration and data access are different.

```
PROGRAM MAIN
VAR
  aPoints : ARRAY[1..2,1..3] OF INT := [1,2,3,4,5,6];
  a2Boxes : ARRAY[1..2] OF ARRAY[1..3] OF INT := [ [1, 2, 3], [4, 5, 6]];
  a3Boxes : ARRAY[1..2] OF ARRAY[1..3] OF ARRAY[1..4] OF INT := [ [ [1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12] ], [ [13, 14, 15, 16], [17, 18, 19, 20], [21, 22, 23, 24] ] ];
  a4Boxes : ARRAY[1..2] OF ARRAY[1..3] OF ARRAY[1..4] OF ARRAY[1..5] OF INT;
END_VAR

aPoints[1, 2] := 1200;
a2Boxes[1][2] := 1200;
```

	aPoints	ARRAY [1..2, 1..3] OF INT	
	aPoints[1, 1]	INT	1
	aPoints[1, 2]	INT	1200
	aPoints[1, 3]	INT	3
	aPoints[2, 1]	INT	4
	aPoints[2, 2]	INT	5
	aPoints[2, 3]	INT	6
	a2Boxes	ARRAY [1..2] OF ARRAY [1..3] OF INT	
	a2Boxes[1]	ARRAY [1..3] OF INT	
	a2Boxes[1][1]	INT	1
	a2Boxes[1][2]	INT	1200
	a2Boxes[1][3]	INT	3
	a2Boxes[2]	ARRAY [1..3] OF INT	
	a2Boxes[2][1]	INT	4
	a2Boxes[2][2]	INT	5
	a2Boxes[2][3]	INT	6

16.5.16.3 Array with variable length

In function blocks, functions or methods, arrays with variable length can be declared in the declaration section VAR_IN_OUT. The operators LOWER_BOUND and UPPER_BOUND can be used to determine the index limits of the array that is actually used at runtime. LOWER_BOUND returns the lower limit, UPPER_BOUND returns the upper limit.



Only statically declared arrays may be passed to an array with variable length. Dynamic arrays created using the __NEW operator must not be passed.

Syntax for declaring a one-dimensional array of variable length

```
<variable name> : ARRAY[*] OF <data type> ( := <initialization> )? ;
<data type> : elementary data types | user defined data types | function block types
// (...)? : Optional
```

Syntax for declaring a multi-dimensional array of variable length

```
<variable name> : ARRAY[* ( , * )+ ] OF <data type> ( := <initialization> )? ;
<data type> : elementary data types | user defined data types | function block types
// (...) + : One or more further dimensions
// (...) ? : Optional
```

Syntax of the operators for limit index calculation

```
LOWER_BOUND( <variable name> , <dimension number> )
UPPER_BOUND( <variable name> , <dimension number> )
```

Example

The function F_SUM adds the integer values of the array elements and returns the calculated sum as the result. The sum is calculated over all array elements present at runtime. Since the actual number of array elements will only be known at runtime, the local variable is declared as a one-dimensional array of variable length. Arrays with different fixed lengths can be passed to this addition function.

```
FUNCTION F_Sum : DINT;
VAR_IN_OUT
  aData      : ARRAY [*] OF INT;
END_VAR
VAR
  i, nSum    : DINT;
END_VAR

nSum := 0;

FOR i := LOWER_BOUND(aData,1) TO UPPER_BOUND(aData,1) DO
  nSum := nSum + aData[i];
END_FOR;

F_Sum := nSum;
```

16.5.17 Structure

A structure is a user-defined data type and combines several variables with any data type into a logical unit. The variables declared within a structure are referred to as components.

You declare a structure in a DUT object that you create in the project using the command **Add > DUT** in the context menu of the PLC project tree.

Syntax:

```
TYPE <structure name> :
STRUCT
  (<variable declaration optional with initialization>)+
END_STRUCT
END_TYPE
```

The <structure name> is an identifier that is valid in the entire project, allowing you to use it like a standard data type. In addition, you can declare as many variables as you like (at least one), which are optionally supplemented by an initialization.

Furthermore, you can nest structures. This means that you declare a structure component with an existing structure type. The only restriction is that you may not assign any addresses to the variables (structure components) (the AT declaration is not permitted here).

Sample: structure declaration

```
TYPE ST_POLYGONLINE :
STRUCT
  aStart : ARRAY[1..2] OF INT := [-99, -99];
  aPoint1 : ARRAY[1..2] OF INT;
  aPoint2 : ARRAY[1..2] OF INT;
  aPoint3 : ARRAY[1..2] OF INT;
```

```

    aPoint4 : ARRAY[1..2] OF INT;
    aEnd : ARRAY[1..2] OF INT := [99, 99];
END_STRUCT
END_TYPE

```

● 8-byte alignment



An 8-byte alignment was introduced with TwinCAT 3. Make sure the alignment is correct if data are exchanged as a complete memory block with other controllers or software components (see [Alignment \[► 791\]](#)).

Extending a type declaration

Starting from an existing structure, a further structure is declared. In addition to its own components, the extended structure has the same structure components as the basic structure.

Syntax

```

TYPE <structure name> EXTENDS <basis structure> :
STRUCT
    (<variable declaration optional with initialization>)+
END_STRUCT
END_TYPE

```

Sample: structure declaration

```

TYPE ST_PENTAGON EXTENDS ST_POLYGONLINE :
STRUCT
    aPoint5 : ARRAY[1..2] OF INT;
END_STRUCT
END_TYPE

```

Declaration and initialization of structures

Sample

```

PROGRAM Line
VAR
    stPolygon : ST_POLYGONLINE := (aStart:=[1,1], aPoint1:=[5,2], aPoint2:=[7,3], aPoint3:=[8,5],
aiPoint4:=[5,7], aEnd:=[1,1]);
    stPentagon : ST_PENTAGON := (aStart:=[0,0], aPoint1:=[1,1], aPoint2:=[2,2], aPoint3:=[3,3],
aPoint4:=[4,4], aPoint5:=[5,5], aEnd:=[0,0]);
END_VAR

```

You cannot use initializations with variables. For an example of how to initialize an array of a structure, see the help page for data type ARRAY.

See also:

- [Object DUT \[► 75\]](#)

Accessing a structure component

You access a structure component according to the following syntax:

```
<variable name>.<component name>
```

Sample

```

PROGRAM Polygon
VAR
    stPolygon : ST_POLYGONLINE := := (aStart:=[1,1], aPoint1:=[5,2], aPoint2:=[7,3], aPoint3:=[8,5],
aiPoint4:=[5,7], aEnd:=[1,1]);
    nPoint : INT;
END_VAR
// Assigns 5 to nPoint
nPoint := stPolygon.aPoint1[1];

```

Ergebnis: nPoint = 5

Symbolic bit access in structure variables

You can declare a structure with variables of the data type BIT in order to combine individual bits into a logical unit. You can then address the individual bits symbolically via a name (instead of via the bit index).

Syntax declaration:

```
TYPE <structure name> :
STRUCT
  ( <bit name> : BIT; )+
END_STRUCT
END_TYPE
```

Bit access syntax:

<Structure name>.<Bit name>

Sample:

Structure declaration

```
TYPE ST_CONTROL :
STRUCT
  bitOperationEnabled : BIT;
  bitSwitchOnActive   : BIT;
  bitEnableOperation  : BIT;
  bitoterror           : BIT;
  bitVoltageEnabled   : BIT;
  bitQuickStop        : BIT;
  bitSwitchOnLocked   : BIT;
  bitWarning           : BIT;
END_STRUCT
END_TYPE
```

Bit access

```
FUNCTION_BLOCK FB_Controller
VAR_INPUT
  bStart : BOOL;
END_VAR
VAR_OUTPUT
END_VAR
VAR
  stControl : ST_CONTROL;
END_VAR

IF bStart = TRUE THEN
// Symbolic bit access
stControl.bitEnableOperation := TRUE;
END_IF

PROGRAM MAIN
VAR
fbController : FB_Controller;
END_VAR

fbController();
fbController.bStart := TRUE;
```



References and pointers to BIT variables are invalid declarations, as are array components with the base type BIT.

See also:

- [BIT \[▶ 759\]](#)
- [ARRAY \[▶ 773\]](#)

Also see about this

- [Object DUT \[▶ 75\]](#)
- [EQ \[▶ 730\]](#)
- [ARRAY \[▶ 773\]](#)

16.5.18 Enumerations

An enumeration is a user-defined data type composed of a comma-separated series of components, also called enumeration values, that is used to declare user-defined variables. In addition, you can use the enumeration components like constant variables, whose identifiers `<enumeration name>.<component name>` are globally known in the project.

You declare an enumeration in a DUT object that you create in the project using the command **Add > DUT** in the context menu of the PLC project tree.

i Each enumeration that you add to a project is automatically given an [attribute 'strict'](#) [▶ 823] and an [attribute 'qualified only'](#) [▶ 822] in the line above the TYPE declaration. The attributes can also be added or removed explicitly.

See also:

- [Object DUT](#) [▶ 75]
- [Attribute 'to string'](#) [▶ 836]
- [TO STRING/TO WSTRING for enumeration variables](#) [▶ 726]

Declaration

Syntax

```
{attribute 'strict'}
TYPE <enumeration name>:
(
  <component declaration>,
  <component declaration>
) <basic data type> := <default variable initialization>
;
END_TYPE
```

{attribute 'strict'}	Optional The pragma causes a strict type check, as described below, to be carried out. The pragma is optional, but recommended.
<enumeration name>	Name of the enumeration that can be used as a data type in the code Example: E_ColorBasic
<component declaration>	Two or above components (any number equal to or above two) The values of the components are initialized automatically: starting at 0, the values are incremented consecutively by 1. However, you can also explicitly assign fixed initial values to the individual components. Example: eYellow := 1
<basic data type>	Optional You can explicitly declare one of the following basic data types: UINT SINT USINT DINT UDINT LINT ULINT BYTE WORD DWORD LWORD If no explicit base data type is declared, the INT base data type is used automatically.
<default variable initialization>	Optional One of the components can be explicitly declared as the initial component. If no explicit initialization is specified, initialization is automatically carried out with the top component.

Example

```
{attribute 'qualified_only'}
{attribute 'strict'}
TYPE E_ColorBasic :
(
  eRed,
```

```

    eYellow,
    eGreen := 10,
    eBlue,
    eBlack
) // Basic data type is INT, default initialization for all E_ColorBasic variables is eYellow
;
END_TYPE

```

In this declaration, the first two components receive the standard initialization values: eRed = 0, eYellow = 1, and the initialization value of the third component is explicitly defined differently: eGreen = 10. The following components then receive standard initialization values: eBlue = 11, eBlack = 12.

Enumeration with explicit base data type

Extensions for the IEC 61131-3 standard

The base data type for an enumeration declaration is INT by default. However, you can also declare enumerations that are explicitly based on a different integer data type.

Example: Enumeration with DWORD basic data type

```

TYPE E_Color :
(
    eWhite   := 16#FFFFFF,
    eYellow  := 16#FFFF00,
    eGreen   := 16#FF00FF,
    eBlue    := 16#0000FF,
    eBlack   := 16#000000
)DWORD := eBlack
; // Basic data type is DWORD, default initialization for all E_Color variables is eBlack
END_TYPE

```

Strict programming rules



From TwinCAT 3.1 Build 4026, the pragma {attribute 'strict'} is automatically added to the first line of an enumeration.

The strict programming rules are activated on adding the pragma {attribute 'strict'}.

The following code is then classified as a compiler error:

- Arithmetical operations with enumeration components For example, an enumeration variable cannot be used as a counter variable in a FOR loop.
- Assignment to an enumeration component of a constant value that does not correspond to an enumeration value.
- Assignment to an enumeration component of a non-constant variable that has a different data type than the enumeration.

Arithmetical operations can lead to undeclared values being assigned to enumeration components. It is better programming style to use SWITCH/CASE instructions in order to process component values.

Declaration and initialization of enumeration variables

Syntax

```
<variable name> : <enumeration name> ( := <initialization> )? ;
```

When declaring an enumeration variable with user-defined data type <enumeration name>, this can be initialized with an enumeration component.

Sample

```

PROGRAM MAIN
VAR
    eColorCar   : E_Color;
    eColorTaxi  : E_Color := E_Color.eYellow;
END_VAR

```

The variable `eColorCar` is initialized with `E_Color.eblack`. This is the standard initialization for all enumeration variables of the type `E_Color` and is thus defined in this way in the type declaration. The variable `eColorTaxi` has its own initialization.

If no initializations are specified, initialization takes place with 0.

```
PROGRAM MAIN
VAR
    eColorFlower : E_ColorBasic;
    eColorTree   : E_ColorBasic := E_ColorBasic.eGreen;
END_VAR
```

The variable `eColorFlower` is initialized with `E_ColorBasic.eYellow`. This is the standard initialization for all enumeration variables of the type `E_ColorBasic`. Since no component for the initialization is specified in the enumeration declaration, initialization automatically takes place with the component that has the value 0. This is normally the first of the enumeration components. However, this can also be another component that is not in first place, but is explicitly initialized with 0. The variable `eColorTree` has an explicit initialization.

If no value is specified for both the type and the variable, the following rule applies: if an enumeration contains a value for 0, then this value is the standard initialization; if not, then the first component in the list.

Samples

Initialization with the 0 component:

```
TYPE E_SampleA :
(
    e1 := 2,
    e2 := 0,
    e3
);
END_TYPE

PROGRAM MAIN
VAR
    eSampleA : E_SampleA;
END_VAR
```

The variable `eSampleA` is initialized with `E_SampleA.e2`.

Initialization with the first component:

```
TYPE E_SampleB :
(
    e1 := 3,
    e2 := 1,
    e3
);
END_TYPE

PROGRAM MAIN
VAR
    eSampleB : E_SampleB;
END_VAR
```

The variable `eSampleB` is initialized with `E_SampleB.e1`.

Extensions of the IEC 61131-3 standard

The enumeration components can also be used as constant variables with the identifier `<enumeration name>.<component name>`. Enumeration components are globally known in the project and access to them is explicit. Therefore, a component name can be used in different enumerations.

Sample: component blue

```
PROGRAM MAIN
VAR
    eFlower   : E_ColorBasic;
    eColorCar : E_Color;
END_VAR

// unambiguous identifiers although the component names are identical
eFlower := E_ColorBasic.eBlue;
eColorCar := E_Color.eBlue;
```

```
// invalid code
eFlower := eBlue;
eColorCar := eBlue;
```

16.5.19 Alias

An alias is a user-defined data type that can be used to create an alternative name for a data type or function block.

You declare an alias in a DUT object that you create using the command **Add > DUT** in the context menu of the PLC project tree in the project.

Syntax:

```
TYPE <identifier> : <Assignment statement>;
END_TYPE
```

Example:

The example shows the declaration of an alias T_Message. A PLC variable of type T_Message declared in the program is always a string with 50 characters.

```
TYPE T_Message : STRING[50];
END_TYPE
```

Declaration:

```
sMessageA : T_Message;
```

Program:

```
sMessageA := 'This is a message';
```

See also:

- [Object DUT \[▶ 75\]](#)

16.5.20 UNION

A UNION is a data structure that usually contains different data types. In a union all components have the same offset, which means they occupy the same memory space.

You declare an union in a DUT object that you create using the command **Add > DUT** in the context menu of the PLC project tree in the project.

Syntax:

```
TYPE <union name>:
UNION
  <Variable declaration 1>
  ...
  <Variable declaration n>
END_UNION
END_TYPE
```

Sample 1:

In the following sample declaration of a union, an assignment to uName.fA also affects uName.nB and uName.nC.

Declaration:

```
TYPE U_Name:
UNION
  fA : LREAL;
  nB : LINT;
  nC : WORD;
END_UNION
END_TYPE
```

Instantiation:

```
VAR
  uName : U_Name;
END_VAR
```

Assignment:

```
uName.fA := 1;
```

Result:

```
fA = 1
nB = 16#3FF0000000000000
nC = 0
```

Sample 2:

In the following sample declaration of a union, an assignment to u2Byte.nUINT also affects u2Byte.a2Byte and u2Byte.aBits.

Declaration of the structure ST_Bits:

```
TYPE ST_Bits :
STRUCT
  bBit1 : BIT;
  bBit2 : BIT;
  bBit3 : BIT;
  bBit4 : BIT;
  bBit5 : BIT;
  bBit6 : BIT;
  bBit7 : BIT;
  bBit8 : BIT;
END_STRUCT
END_TYPE
```

Declaration of the union U_2Byte:

```
TYPE U_2Byte :
UNION
  nUINT : UINT;
  a2Byte : ARRAY[1..2] OF BYTE;
  aBits : ARRAY[1..2] OF ST_Bits;
END_UNION
END_TYPE
```

Instantiation of the union:

```
VAR
  u2Byte : U_2Byte;
END_VAR
```

Assignment 1:

```
u2Byte.nUINT := 5;
```

Result 1:

Expression	Type	Value
u2Byte	U_2Byte	
nUINT	UINT	5
a2Byte	ARRAY [1..2] OF BYTE	
a2Byte[1]	BYTE	5
a2Byte[2]	BYTE	0
aBits	ARRAY [1..2] OF ST_Bits	
aBits[1]	ST_Bits	
bBit1	BIT	TRUE
bBit2	BIT	FALSE
bBit3	BIT	TRUE
bBit4	BIT	FALSE
bBit5	BIT	FALSE
bBit6	BIT	FALSE
bBit7	BIT	FALSE
bBit8	BIT	FALSE
aBits[2]	ST_Bits	
bBit1	BIT	FALSE
bBit2	BIT	FALSE
bBit3	BIT	FALSE
bBit4	BIT	FALSE
bBit5	BIT	FALSE
bBit6	BIT	FALSE
bBit7	BIT	FALSE
bBit8	BIT	FALSE

Assignment 2:

```
u2Byte.nUINT := 255;
```

Result 2:

Expression	Type	Value
[-] u2Byte	U_2Byte	
[-] nUINT	UINT	255
[-] a2Byte	ARRAY [1..2] OF BYTE	
[-] a2Byte[1]	BYTE	255
[-] a2Byte[2]	BYTE	0
[-] aBits	ARRAY [1..2] OF ST_Bits	
[-] aBits[1]	ST_Bits	
[-] bBit1	BIT	TRUE
[-] bBit2	BIT	TRUE
[-] bBit3	BIT	TRUE
[-] bBit4	BIT	TRUE
[-] bBit5	BIT	TRUE
[-] bBit6	BIT	TRUE
[-] bBit7	BIT	TRUE
[-] bBit8	BIT	TRUE
[-] aBits[2]	ST_Bits	
[-] bBit1	BIT	FALSE
[-] bBit2	BIT	FALSE
[-] bBit3	BIT	FALSE
[-] bBit4	BIT	FALSE
[-] bBit5	BIT	FALSE
[-] bBit6	BIT	FALSE
[-] bBit7	BIT	FALSE
[-] bBit8	BIT	FALSE

Assignment 3:

```
u2Byte.nUINT := 256;
```

Result 3:

Expression	Type	Value
u2Byte	U_2Byte	
nUINT	UINT	256
a2Byte	ARRAY [1..2] OF BYTE	
a2Byte[1]	BYTE	0
a2Byte[2]	BYTE	1
aBits	ARRAY [1..2] OF ST_Bits	
aBits[1]	ST_Bits	
bBit1	BIT	FALSE
bBit2	BIT	FALSE
bBit3	BIT	FALSE
bBit4	BIT	FALSE
bBit5	BIT	FALSE
bBit6	BIT	FALSE
bBit7	BIT	FALSE
bBit8	BIT	FALSE
aBits[2]	ST_Bits	
bBit1	BIT	TRUE
bBit2	BIT	FALSE
bBit3	BIT	FALSE
bBit4	BIT	FALSE
bBit5	BIT	FALSE
bBit6	BIT	FALSE
bBit7	BIT	FALSE
bBit8	BIT	FALSE

See also:

- [Object DUT \[▶ 75\]](#)

16.6 Global data types

16.6.1 Overview

TwinCAT 3 provides a type system for the management of data types. The type system consists of system basic types and can be extended by custom data types through the customer project. (See also [TwinCAT 3 Type system](#))

In this section you will find a description of global data types provided by the TwinCAT system for the PLC:

- [PlcAppSystemInfo \[▶ 788\]](#)
- [PlcTaskSystemInfo \[▶ 790\]](#)
- [ST_LibVersion \[▶ 791\]](#)

16.6.2 PlcAppSystemInfo

Each PLC contains an instance of type 'PlcAppSystemInfo' with the name '_AppInfo'.

The corresponding namespace is 'TwinCAT_SystemInfoVarList'. This must be specified for use in a library, for example.

```

TYPE PlcAppSystemInfo
STRUCT
  ObjId          : OTCID;
  TaskCnt       : UDINT;

```



```

OnlineChangeCnt : UDINT;
Flags           : DWORD;
AdsPort        : UINT;
BootDataLoaded : BOOL;
OldBootData    : BOOL;
AppTimestamp   : DT;
KeepOutputsOnBP : BOOL;
ShutdownInProgress : BOOL;
LicensesPending : BOOL;
BSODOccured   : BOOL;
LoggedIn       : BOOL;
PersistentStatus : EPlcPersistentStatus;

TComSrvPtr     : IComObjectServer;

AppName        : STRING(63);
ProjectName    : STRING(63);
END_STRUCT
END_TYPE

```

ObjId	Object ID of the PLC project instance
TaskCnt	Number of tasks in the runtime system
OnlineChangeCnt	Number of online changes since the last complete download
Flags	Reserved
AdsPort	ADS port of the PLC application
BootDataLoaded	PERSISTENT variables: LOADED (without error)
OldBootData	PERSISTENT variables: INVALID (the back-up copy was loaded, since no valid file was present)
AppTimestamp	Time at which the PLC application was compiled
KeepOutputsOnBP	The flag can be set and prevents that the outputs are zeroed when a breakpoint is reached. In this case the task continues to run. Only the execution of the PLC code is interrupted.
ShutdownInProgress	This variable has the value TRUE if a shutdown of the TwinCAT system is in progress. Some parts of the TwinCAT system may already have been shut down.
LicensesPending	This variable has the value TRUE if not all licenses that are provided by license dongles have been validated yet.
BSODOccured	This variable has the value TRUE, if Windows is in a BSOD.
LoggedIn	This variable has the value TRUE if at least one XAE instance is logged into the project.
PersistentStatus	PERSISTENT variables: Restoration status: PS_None: No persistent variables were restored. PS_All: All variables that are persistent in the current project state were restored. PS_Partial: Fewer variables were restored than are persistent in the current project state.
TComSrvPtr	Pointer to the TcCOM object server
AppName	Name generated by TwinCAT, which contains the port.
ProjectName	Name of the project

Differences compared with TwinCAT 2

If the variable runTimeNo was used under TwinCAT 2, the corresponding program code must be converted for application under TwinCAT 3.

Example:

- **Application under TwinCAT 2:** nPlcAdsPort := 801 + (SystemInfo.runTimeNo - 1) * 10;
- **Application under TwinCAT 3:** nPlcAdsPort := _AppInfo.AdsPort;

16.6.3 PlcTaskSystemInfo

Each PLC contains an array of instances of this type. The name of the arrays is '_TaskInfo[]'. The individual instances of this type can be accessed by using the index of the corresponding task as array index. The task index can be read out using the GETCURTASKINDEXEX function, which is provided by the Tc2_System library.

The corresponding namespace is 'TwinCAT_SystemInfoVarList'. This must be specified for use in a library, for example.

```
{attribute 'Namespace' := 'PLC'}
TYPE PlcTaskSystemInfo
STRUCT
    ObjId          : OTCID;
    CycleTime      : UDINT;
    Priority        : UINT;
    AdsPort        : UINT;
    CycleCount     : UDINT;
    DcTaskTime     : LINT;
    LastExecTime   : UDINT;
    FirstCycle     : BOOL;
    CycleTimeExceeded : BOOL;
    InCallAfterOutputUpdate : BOOL;
    RTViolation    : BOOL;

    TaskName      : STRING(63);
END_STRUCT
END_TYPE
```

ObjId	Object ID of the task reference, from which the PLC program is called.
CycleTime	Set task cycle time in multiples of 100 ns
Priority	Set task priority
AdsPort	ADS port of the task
CycleCount	Cycle counter Please note that the cycle counter refers to the actual system task and not to the task reference of the PLC project. The background to this is that several PLC projects or other TcCOM modules can share a task and these modules can thus access the same cycle counter. As a result, the cycle counter is reset during a TwinCAT restart in RUN mode (and not when resetting the PLC project).
DcTaskTime	Distributed Clock System Time. It remains constant for a task runtime.
LastExecTime	Cycle time required for the last cycle in multiples of 100 ns
FirstCycle	During the first PLC task cycle, this variable has the value TRUE.
CycleTimeExceeded	This variable indicates whether the set task cycle time was exceeded. The value of the CycleTimeExceeded variable is updated in each cycle. The variable is set to 0 if the previous cycle has been completed within its intended timeframe, otherwise to 1.
InCallAfterOutputUpdate	This variable has the value TRUE if the origin of the current call is declared with the attribute 'TcCallAfterOutputUpdate'.
RTViolation	This variable has the value TRUE if the real-time limit is exceeded on a mixed core (Windows + real-time on one core). In this case the value TRUE refers to the core on which the corresponding task is running.
TaskName	Name of the task reference object in the PLC project (e.g. 'MyPlcProject_PlcTaskRef')

Sample:

```
VAR
    nTaskIdx      : DINT;
    nCycleTime    : UDINT;
END_VAR

nTaskIdx := GETCURTASKINDEXEX();
IF nTaskIdx > 0 THEN
    nCycleTime := _TaskInfo[nTaskIdx].CycleTime;
END_IF
```

16.6.4 ST_LibVersion

This structure defines the version of a PLC library. Each library contains a globally declared instance of this data type.

```

TYPE ST_LibVersion :
STRUCT
    iMajor      : UINT;
    iMinor      : UINT;
    iBuild      : UINT;
    iRevision   : UINT;
    nFlags      : DWORD
    sVersion    : STRING(23);
END_STRUCT
END_TYPE
    
```

iMajor	Major number
iMinor	Minor number
iBuild	Build number
iRevision	Revision number
nFlags	Enabling library nFlags = 1, the library is enabled (check mark set), nFlags = 0, the library is not enabled.
sVersion	Complete version as string

16.7 Alignment

An 8-byte alignment was introduced with TwinCAT 3.

System	Alignment
TwinCAT 2, x86	1 byte
TwinCAT 2, ARM	4 bytes
TwinCAT 3	8 bytes

Particular attention is required if data are exchanged as a whole memory block with other controllers or software components such as visualizations.

Memory location

- A variable is located at the memory location that is specified by the size of its data type and by the alignment.
 - If the data type is smaller than or equal to the alignment of the system, the memory location of the variable is equivalent to a multiple of the data type size.
 - If the data type is larger than the alignment of the system, the memory location of the variable is equivalent to a multiple of the alignment.
- A structured data type (structure, function block) is located at the memory location that is specified by the size of the largest data type in the structure and by the alignment.
 - If the largest data type is smaller than or equal to the alignment of the system, the memory location of the structure instance is equivalent to a multiple of this data type size.
 - If the largest data type is larger than the alignment of the system, the memory location of the structure instance is equivalent to a multiple of the alignment.

Memory space

- From the above memory location a variable takes up as much memory as is determined by the size of its data type.
- A structured data type (structure, function block) takes up the amount of memory space, starting from the above memory location, as is specified by the size of the largest data type in the structure and by the alignment.

- If the largest data type is smaller than or equal to the alignment of the system, the memory space taken up by the structure instance is equivalent to a multiple of this data type size.
- If the largest data type is larger than the alignment of the system, the memory space taken up by the structure instance is equivalent to a multiple of the alignment.

These alignment rules may result in implicit padding bytes.



Please note (e.g. for a memory comparison with the help of the Tc2 system function MEMCMP), that padding bytes are not initialized.



Largest data type in a function block

Each function block implicitly contains at least one pointer to store the address of the virtual method table. This means that in a target system with 64-bit architecture, the function block contains a data type with a size of 8 bytes. Depending on the function block, this can be the largest data type it contains, which determines the memory location and memory space.

Sample 1 (TwinCAT 3)

```

TYPE ST_Test1 :
STRUCT
  fVar   : LREAL; // 8 Byte
  nVar1  : DINT;  // 4 Byte
  nVar2  : SINT;  // 1 Byte
  (* 3 filler bytes to reach a limit corresponding to the largest data type (divisible by 8 byte)
*)
END_STRUCT
END_TYPE

TYPE ST_Test2 :
STRUCT
  nVar2  : SINT;  // 1 Byte
  (* 3 filler bytes to reach a limit corresponding to the following data type (divisible by 4 byte)
*)
  nVar1  : DINT;  // 4 Byte
  fVar   : LREAL; // 8 Byte
END_STRUCT
END_TYPE

TYPE ST_Test3 :
STRUCT
  nVar2  : SINT;  // 1 Byte
  (* 7 filler bytes to reach a limit corresponding to the following data type (divisible by 8 byte)
*)
  fVar   : LREAL; // 8 Byte
  nVar1  : DINT;  // 4 Byte
  (* 4 filler bytes to reach a limit corresponding to the largest data type (divisible by 8 byte)
*)
END_STRUCT
END_TYPE

```

Due to the 8-byte alignment of TwinCAT 3, implicit padding bytes are added. The overall size of the structures can vary, although they contain three variables of the same data type.

Expression	Type	Value
stTest1	ST_Test1	
stTest2	ST_Test2	
stTest3	ST_Test3	
nSize1	UDINT	16
nSize2	UDINT	16
nSize3	UDINT	24

```

1 nSize1 16 :=SIZEOF(stTest1);
2 nSize2 16 :=SIZEOF(stTest2);
3 nSize3 24 :=SIZEOF(stTest3);
4 RETURN

```

On a system with 1-byte alignment the three structures would have the same memory requirement of 13 bytes.

Sample 2

```

TYPE ST_Test :
STRUCT
  nDWORD : DWORD;    // 4 Byte
  (* With an 8-byte alignment: 4 filler bytes *)
  nLWORD : LWORD;    // 8 Byte
END_STRUCT
END_TYPE
    
```

With a 4-byte alignment the structure has a size of 12 bytes. No padding bytes are inserted. Reason:

- The variable nLWORD is located in a memory location that is equivalent to a multiple of the alignment, since its data type (8 bytes) is larger than the alignment (4 bytes).
- The structured data type takes up an amount of memory space equivalent to a multiple of the alignment, since the largest data type in the structure (8 bytes) is larger than the alignment (4 bytes).

With an 8-byte alignment, conversely, the structure has a size of 16 bytes, since 4 padding bytes are inserted between the two variables. Reason:

- The variable nLWORD is located in a memory location that is equivalent to a multiple of the size of its data type, since its data type (8 bytes) is smaller than or equal to the alignment (8 bytes).
- The structured data type takes up an amount of memory space equivalent to a multiple of the size of the largest data type, since the largest data type in the structure (8 bytes) is smaller than or equal to the alignment (8 bytes).

Further information & sample project:

- Refer also to the [Attribute 'pack mode' \[► 816\]](#) in order to define how a data structure should be packed.
- On the [PLC Samples page, \[► 1130\]](#) you can also download a sample project on the topic of byte alignment.

16.8 Pragmas




Pragma instructions affect the properties of one or more variables regarding the compilation or precompilation process. Various categories of pragmas are available for this purpose.

16.8.1 Message pragmas

Message pragmas are used to force the output of messages in the message window during the compilation process.

The pragma instruction can be inserted in a separate or an existing line in the text editor of a POU.

Types

Pragma	Message type
{text <'textstring'>}	Text: Output of <text string>.
{info <'textstring'>}	 Information: Output of <text string>.
{warning <'textstring'>}	 Warning: Output of <text string>. In contrast to the attribute pragma 'obsolete', you define the warning locally for the current position.
{error <'textstring'>}	 Error: Output of <text string>.



In the TwinCAT message window, you can access the source position of a message of category **Information**, **Warning** and **Error** by using the commands **Next Message** or **Previous Message**. This means you get to the position where the pragma is added in the source code.

Example:

```
VAR
  nVar : INT; {info 'TODO: should get another name'}
  bVar : BOOL;
  aTest : ARRAY [0..10] OF INT;
  nIdx : INT;
END_VAR

aTest[nIdx] := aTest[nIdx]+1;
nVar := nVar+1;

{warning 'This is a warning'}
{text 'Part xy has been compiled completely'}
```

Output in the message window:

	Description	File	Line	Column	Project
1	----- Build started: Application: TwinCAT_Device.Project12 -----		0	0	
2	typify code ...		0	0	
3	generate code...		0	0	
4	generate global initializations ...		0	0	
5	generate code initialization ...		0	0	
6	generate relocations ...		0	0	
7	TODO: should get another name	MAIN.TcPOU	3	1	Project12
8	This is a warning	MAIN.TcPOU	4	1	Project12
9	Part xy has been compiled completely	MAIN.TcPOU	5	1	Project12
10	Size of generated code: 23980 bytes		0	0	
11	Size of global data: 5307 bytes		0	0	

See also:

- [Conditional pragmas \[▶ 837\]](#)

16.8.2 Attribute pragmas

Attribute pragmas are used to influence compilation and precompilation.

TwinCAT supports a number of predefined attribute pragmas. In addition, you can use custom pragmas, which you can query by using conditional pragmas before compiling the project.



If you define your own attributes, make sure they are unambiguous. You can achieve this by assigning a prefix to the attribute name, for example.

Attributes are defined in the declaration part. Exception: You can only use attributes in **Action** or **Transition** objects if the implementation language of the action or transition is “Structured Text (ST)”. Since actions and transitions do not have a declaration part, you define attributes at the start of the implementation part.

16.8.2.1 User-defined attributes

Custom attributes are any project or user-defined attributes that you can apply to POU, actions, data type definitions and variables. You can query a custom attribute before compiling the PLC project using conditional pragmas.



You can query custom attributes with conditional pragmas using the `hasattribute` operator.

Syntax: {attribute 'attribute'}

Examples of POU and actions:

Attribute 'vision' for function F_Sample:

```
{attribute 'vision'}
FUNCTION F_Sample : INT
VAR_INPUT
    nSample : INT;
END_VAR
```

Examples for variables:

Attribute 'DoCount' for variable nVar:

```
PROGRAM MAIN
VAR
    {attribute 'DoCount'};
    nVar : INT;
    bVar : BOOL;
END_VAR
```

Example for data types:

Attribute 'aType' for data type ST_MyType:

```
{attribute 'aType'}
TYPE ST_MyType :
STRUCT
    nTest : INT;
    bTest : BOOL;
END_STRUCT
END_TYPE
```

See also:

- [Conditional pragmas \[► 837\]](#)

16.8.2.2 Attribute 'c++_compatible'

The pragma causes the VTable generated by the PLC compiler to be binary compatible with that of a C++ compiler. This makes it possible to access the interface methods implemented in the PLC from a TcCom module implemented in C++.

Syntax: {attribute 'c++_compatible'}

Insertion location:

The pragma must be added in the following places:

- Line above the interface declaration
- Line above the declaration of the individual interface methods
- Line above the declaration of the function block that implements the interface
- Line above the declaration of the methods that are implemented from the interface

Sample:

Declaration of a function block that implements a C++-compatible interface:

```
{attribute 'c++_compatible'}
FUNCTION_BLOCK FB_Sample IMPLEMENTS I_Sample
VAR_INPUT
END_VAR
VAR_OUTPUT
END_VAR
VAR
END_VAR
```

Declaration of a method that is defined in the interface:

```
{attribute 'c++_compatible'}
{attribute 'minimal_input_size' := '4'}
{attribute 'pack_mode' := '4'}
```

```
METHOD Method1 : HRESULT
VAR_INPUT
    nParameter1 : INT;
END_VAR
```

See also:

- [Attribute 'minimal input size' \[► 810\]](#)
- [Attribute 'pack mode' \[► 816\]](#)

16.8.2.3 Attribute 'call_after_global_init_slot'

The pragma causes all functions and programs that contain this attribute to be called after the global initialization. Use the attribute value to determine the call sequence.

Syntax: {attribute 'call_after_global_init_slot' := '<slot>'}

<slot> : Integer value that defines the ranking value in the order of the calls: The lower the value, the earlier the call will be. If several function blocks have the same value for the attribute, the order of their calls remains undefined.

Insertion location: First line above the declaration part of functions and programs

If a method has the attribute, TwinCAT determines all instances of the corresponding function block and calls all instances in the specified slot. In this case, you have no influence on the order of the instances.

See also:

- [Methods FB_init, FB_reinit and FB_exit \[► 848\]](#)

16.8.2.4 Attribute 'call_after_init'

The pragma causes a method to be called implicitly after the initialization of a function block instance. Strictly speaking, the method is called after the initial assignments have been processed and before the tasks of a PLC project have been started. It can therefore respond appropriately to the user's specifications. The name of the method is freely selectable (exceptions: FB_init, FB_reinit and FB_exit).

For performance reasons, you must add the attribute to both the function block and the method in a dedicated first line above the declaration part.

Syntax: {attribute 'call_after_init'}

Insertion location: First line above the declaration part of the method and function block

TwinCAT calls the method after the method FB_init, and after the variable values of an initialization expression have become valid in the instance declaration.

call_after_init for derived function blocks

A function block that extends another function block, which uses the attribute 'call_after_init', must also be assigned the attribute.

For reasons of clarity, it is recommended that you overwrite the corresponding method with the same name, signature, and attribute. This requires a call of `SUPER^.MyInit`.



Methods that contain the attribute 'call_after_init' must not have any inputs (VAR_INPUT). TwinCAT issues a corresponding compile error.

**Debugging**

Finding errors in methods that are declared with {attribute 'call_after_init'} is cumbersome because setting breakpoints cannot have the desired effect, for example.

Example:

Definition:


```
{attribute 'call_after_init'}
FUNCTION_BLOCK FB_Sample
... <functionblock definition>

{attribute 'call_after_init'}
METHOD CallAfterInit
... <method definition>
```

The definition implements the following declaration in the following code processing, for example:

```
fbSample : FB_Sample := (nValue1 := 99);
```

Code processing:

```
fbSample.FB_Init();
fbSample.nValue1 := 99;
fbSample.CallAfterInit();
```

In this way, CallAfterInit() is able to respond to the user-defined initialization.

See also:

- [Methods FB_init, FB_reinit and FB_exit \[► 848\]](#)

16.8.2.5 Attribute 'call_after_online_change_slot'

The pragma causes all functions and programs that contain this attribute to be called after an online change. Use the attribute value <slot> to determine the call sequence.

Syntax: {attribute 'call_after_online_change_slot' := '<slot>'}

<slot>: Integer value that defines the ranking value in the order of the calls: The lower the value, the earlier the call will be. If several function blocks have the same value for the attribute, the order of their calls remains undefined.

Insertion location: First line above the declaration part of functions and programs

If a method has the attribute, then TwinCAT determines all instances of the relevant function block. TwinCAT calls all instances in the specified slot. In this case, you have no influence on the order of the instances.



Since the PLC program cannot run during the online change, any code executed in this situation can cause jitter. Therefore, keep the amount of code to be executed as small as possible.

See also:

- TC3 User Interface documentation: [Command Online Change \[► 955\]](#)

16.8.2.6 Attribute 'call_on_type_change'

The pragma is used to identify a method of a function block A, which is to be called as soon as the data type of a function block B, C, ... referenced by A changes. The referencing can be defined by a pointer variable or by a variable of type REFERENCE. You define the function blocks referenced by A whose type change is to trigger the method call in the attribute value.

Syntax:

```
{attribute 'call_on_type_change' := '<name of the first referenced function block>, <name of the second referenced function block>, <name of the ... referenced function block>'}
```

Insertion location: Line above the first line in the declaration of the method

Example:

Function block with references:

```
FUNCTION_BLOCK FB_A
...
VAR
```

```
pVar   : POINTER TO FB_B;
refVar : REFERENCE TO FB_C;
END_VAR
```

Method for responding to a type change in the references FB_B and FB_C:

```
{attribute 'call_on_type_change' := 'FB_B, FB_C'}
METHOD METH_react_on_type_change : INT
VAR_INPUT
```

16.8.2.7 Attribute 'conditionalshow'

The pragma is relevant for function blocks and other types contained in a library.

The effect of the pragma is that the complete function block containing this attribute is not displayed on the user interface if the associated library is installed as a *.compiled library. However, the function block is displayed on the user interface if the associated library is installed as a *.library.

Affected features:

- Library management
- Debugging
- Input assistant
- List components function
- Monitoring
- Symbol configuration

This is useful when you are developing libraries. As a library developer, you provide function blocks or variables with the pragma. This allows you to specify which identifiers are hidden after being included in a project. If you miss the hidden identifiers later, for example when debugging or further developing the library, you can re-enable their visibility.

Syntax: {attribute 'conditionalshow'}

Insertion location: Line above a signature

Samples

Hide a variable:

```
FUNCTION_BLOCK FB_Sample
PROGRAM_MAIN
VAR
{attribute 'conditionalshow'}
    nLocal   : INT;
    nCounter : INT;
END_VAR
```

The variable nLocal is invisible.

Hide a function block:

```
{attribute 'conditionalshow'}
FUNCTION_BLOCK FB_Sample
VAR
    nLocal   : INT;
    nCounter : INT;
END_VAR
```

The function block FB_Sample and its variables nLocal and nCounter are invisible.

See also:

- [Attribute 'conditionalshow_all_locals' \[► 799\]](#)
- [Attribute 'hide' \[► 805\]](#)
- [Attribute 'hide_all_locals' \[► 806\]](#)

16.8.2.8 Attribute 'conditionalshow_all_locals'

The pragma is relevant for function blocks and other types contained in a library.

The effect of the pragma is that no local variables are displayed on the user interface if the associated library is installed as a *.compiled library. However, the local variables are displayed on the user interface if the associated library is installed as a *.library.

Affected features:

- Library management
- Debugging
- Input assistant
- **List components** function
- Monitoring
- Symbol configuration

This is useful when you are developing libraries. As a library developer, you provide function blocks or variables with the pragma. This allows you to specify which identifiers are hidden after being included in a project. If you miss the hidden identifiers later, for example when debugging or further developing the library, you can re-enable their visibility.

Syntax: {attribute 'conditionalshow_all_locals'}

Insertion location: Line above a signature

Sample: Hide all variables

```
{attribute 'conditionalshow_all_locals'}
FUNCTION_BLOCK FB_Sample
VAR
    nLocal      : INT;
    nCounter    : INT;
END_VAR
```

The local variables nLocal and nCounter of the function block FB_Sample are invisible.

See also:

- [Attribute 'conditionalshow'](#) [► 798]
- [Attribute 'hide'](#) [► 805]
- [Attribute 'hide all locals'](#) [► 806]

16.8.2.9 Attribute 'const_replaced', attribute 'const_non_replaced'

The {attribute 'const_non_replaced'} attribute causes the constant to be replaced in the code, regardless of the setting of the compiler option **Replace constants**. The attribute has an effect on variables of scalar types, but not on composite types such as arrays and structures.

Insert the pragma {attribute 'const_non_replaced'} to explicitly disable the Replace Constants compiler option. This may be desirable so that a constant is available in the symbol configuration, for example.

Syntax:

```
{attribute 'const_replaced'}
```

```
{attribute 'const_non_replaced'}
```

Insertion location: Line above the declaration line of the global variable.

Sample:

The constants nTestCon and bTestCon are available in the symbol configuration because the Replace constants option is disabled.

```

VAR_GLOBAL CONSTANT
  {attribute 'const_non_replaced'}
  nTestCon : INT := 12;
  {attribute 'const_non_replaced'}
  bTestCon : BOOL := TRUE;
  fTestCon : REAL := 1.5;
END_VAR

VAR_GLOBAL
  nTestVar : INT := 12;
  bTestVar : BOOL := TRUE;
END_VAR

```

See also:

- Documentation TC3 User Interface: Properties command (Project) > [Category Compile](#) |> 910]

16.8.2.10 Attribute 'dataflow'

The pragma can be used to control the data flow during processing of function blocks in the FBD/LD/IL editor. The attribute determines at which input or output of a function block the next or previous function block is connected.

You can only assign the attribute to one input and one output in the declaration of a function block.

Syntax: {attribute 'dataflow'}

Insertion location: Line above the declaration line of a variable.

For function blocks without the 'dataflow' attribute, TwinCAT determines the data flow as follows: First, the connection between an output and an input of the same data type is placed. The first input or output variable of the function block is always used first. If there are no variables with the same data type, TwinCAT connects the uppermost output to the uppermost input of the neighboring function blocks.

Example:

The connection between FB and the preceding function block is made via the input variable i1. The connection between FB and the subsequent function block is made via the output variable outRes1.

```

FUNCTION_BLOCK FB
VAR_INPUT
  r1 : REAL;
  {attribute 'dataflow'}
  i1 : INT;
  i2 : INT;
  r2 : REAL;
END_VAR
VAR_OUTPUT
  {attribute 'dataflow'}
  outRes1 : REAL;
  out1 : INT;
  g1 : INT;
  g2 : REAL;
END_VAR

```

See also:

- [FBD/LD/IL](#) |> 108]

16.8.2.11 Attribute 'displaymode'

The pragma defines the display mode of an individual variable. This setting overwrites the global setting for the display of the monitoring variables, which is defined by the commands in the menu

Debug > Representation.

Syntax: {attribute 'displaymode':=<displaymode>}

The following definitions are possible:

- binary format
 - {attribute 'displaymode':='bin'}

- {attribute 'displaymode':='binary'}
- decimal format
 - {attribute 'displaymode':='dec'}
 - {attribute 'displaymode':='decimal'}
- hexadecimal format
 - {attribute 'displaymode':='hex'}
 - {attribute 'displaymode':='hexadecimal'}

Insertion location: Line above the declaration line of a variable

Example:

```
VAR
  {attribute 'displaymode':='hex'}
  nVar1 : DWORD;
END_VAR
```

See also:

- TC3 User Interface documentation: [Command Display Mode - Binary, Decimal, Hexadecimal](#) [► 963]

16.8.2.12 Attribute 'enable_dynamic_creation'

The pragma is required in order to use the `__NEW` operator with function blocks or DUTs.

Syntax: {attribute 'enable_dynamic_creation'}

Insertion location: First line above the declaration part of the function block or DUT

See also:

- Reference Programming: [__NEW](#) [► 732]

16.8.2.13 Attribute 'estimated-stack-usage'

The pragma transfers an estimated value for the stack size requirement.

Methods with recursive calls do not withstand a stack check because the stack usage cannot be determined. Consequently, a warning is issued for these methods. To suppress the warning, you can use the attribute to give the method an estimated value in bytes for the stack size requirement. The method then successfully passes the stack check.

Syntax: {attribute 'estimated-stack-usage' := '<estimated stack size in bytes>'}

Insertion location: First line above the declaration part of the method

Sample:

```
{attribute 'estimated-stack-usage' := '99'} // 99 bytes
METHOD SampleMethod : INT
VAR_INPUT
END_VAR
```

Warning issued:

The warning that is issued for recursive methods without the 'estimated-stack-usage' attribute is:

"C0298: Calculation of stack usage incomplete due to recursive calls, starting at '<FB.Method>'"

In a project that is unchanged compared to the last time the project was build, the warning is only issued when the Rebuild project command is used (not when the Build project command is used).

Setting the stack size:

The value of the maximum possible stack size can be configured in the real-time settings of the TwinCAT project (Settings tab > Global Task Config/Maximum Stack Size [KB]). The set maximum value is not completely available to the PLC project, but part of it is used by the TwinCAT runtime system, for example.

i No stack calculation for stand-alone PLC projects

Due to the decoupling from the System Manager, it is not possible to calculate the stack usage for a stand-alone PLC project.

Recursive method call

Within the associated implementation, a method can call itself: either directly with the help of the THIS pointer or with the help of a local variable for the assigned function block.

NOTICE

Machine downtime due to possible stack overflow

Unexpectedly deep recursions can lead to a stack overflow and thus to a machine downtime.

- Recursions are primarily used for processing recursive data types such as linked lists.

Sample: Calculation of the factorial

Within the function block `FB_Factorial`, the factorial of a number is calculated in different ways. The various calculations are each carried out using a separate method.

- Method `Iterative`: Iterative
- Method `Pragmaed`: Recursive with warning suppression
- Method `Recursive`: Recursive

When the project is rebuilt, only the method `Recursive` generates the warning C0298.

Structure `ST_FactorialResult` for saving the values:

```
// Contains the data of the factorial calculation of nNumber.
TYPE ST_FactorialResult :
STRUCT
  nNumber      : USINT;
  nIterative   : UDINT;
  nRecursive   : UDINT;
  nPragmaed    : UDINT;
END_STRUCT
END_TYPE
```

Function block `FB_Factorial` for calculating the factorial:

```
// Factorial calculation in different ways
FUNCTION_BLOCK FB_Factorial
VAR
  nNumberIterative : UINT;
END_VAR
```

Property `nNr` including set function, for transferring the parameter for the iterative method:

```
{attribute 'monitoring' := 'variable'}
PROPERTY nNr : UINT
nNumberIterative := nNr;
```

Iterative calculation method:

```
// Iterative calculation
METHOD PUBLIC Iterative : UDINT
VAR
  nCnt : UINT;
END_VAR

Iterative := 1;

IF nNumberIterative > 1 AND nNumberIterative <= 12 THEN
  FOR nCnt := 1 TO nNumberIterative DO
    Iterative := Iterative * nCnt;
  END_FOR;
  RETURN;
ELSE
  RETURN;
END_IF
```

Recursive calculation method with attribute for suppressing warning C0298:

```
// Recursive calculation with suppressed warning
{attribute 'estimated-stack-usage' := '200'}
METHOD PUBLIC Pragmaed : UDINT
VAR_INPUT
  nNumber : USINT;
END_VAR

Pragmaed := 1;

IF nNumber > 1 AND nNumber <= 12 THEN
  Pragmaed := nNumber * THIS^.Pragmaed(nNumber := (nNumber - 1));
  RETURN;
ELSE
  RETURN;
END_IF
```

The pragma parameter for specifying the estimated stack size requirement is set to 200 bytes as an example. This is based on the following calculation:

Stack usage per method call:

Return value UDINT + input parameter USINT + method pointer = 4 bytes + 1 byte + 8 bytes = 13 bytes

Stack usage with a maximum of 12 calls:

12 * 13 bytes = 156 bytes, rounded up to 200 bytes

Recursive calculation method:

```
// Recursive calculation
METHOD PUBLIC Recursive : UDINT
VAR_INPUT
  nNumber : USINT;
END_VAR

Recursive := 1;
IF nNumber > 1 AND nNumber <= 12 THEN
  Recursive := nNumber * THIS^.Recursive(nNumber := (nNumber - 1) );
  RETURN;
ELSE
  RETURN;
END_IF
```

Main program MAIN:

```
PROGRAM MAIN
VAR
  fbFactorial : FB_Factorial;
  stFactorial : ST_FactorialResult := (nNumber := 9);
END_VAR

// Iterativ
fbFactorial.nNr      := stFactorial.nNumber;
stFactorial.nIterative := fbFactorial.Iterative();

// Recursive
stFactorial.nRecursive := fbFactorial.Recursive(nNumber := stFactorial.nNumber);

// Pragmaed
stFactorial.nPragmaed := fbFactorial.Pragmaed(nNumber := stFactorial.nNumber);
```

16.8.2.14 Attribute 'ExpandFully'

The pragma causes the components of an array, which is used as input variable for referenced visualizations, to be visualized in the visualization properties dialog.

Syntax: {attribute 'ExpandFully'}

Insertion location: Line above the declaration line of the array

Example:

Visualization visu is to be inserted in a frame within the visualization visu_main. aSample is defined as input variable in the interface editor of visu and is will therefore be available later for assignments in the properties dialog of the frame in visu_main. To also have the individual components of aSample available in this properties dialog, you must insert the 'ExpandFully' attribute in the interface editor of visu directly before aSample. Declaration in the interface editor of visu:

```

VAR_INPUT
{attribute 'ExpandFully'}
aSample : ARRAY[0..5] OF INT;
END_VAR

```

Property	Value
Elementname	GenElemInst_1
Type of element	Frame
Clipping	<input type="checkbox"/>
Show frame	No frame
Scaling type	Anisotropic
Deactivate the background dr...	<input type="checkbox"/>
References	Configure...
Visualization	
aSample	
[0]	
[1]	
[2]	
[3]	
[4]	
[5]	
Position	
Center	
Colors	
Element look	
Texts	

The visualization(s) that are be displayed by this element

16.8.2.15 Attribute 'global_init_slot'

The pragma defines the order of the initialization of programming blocks and global variable lists. Variables within a GVL or POU are initialized from top to bottom. In the case of several global variable lists, the initialization sequence is undefined.

For initializations with literal values, for example 1, 'hello', 3.6, or constants of basic data types, the order of the initializations is irrelevant. However, if there are interdependencies in the initializations, you must set the initialization order. To do this, you can assign a defined initialization slot to a GVL or POU with the attribute 'global_init_slot'.

Constants are initialized before the variables, in the same order as the variables. During initialization, the signatures (function blocks, GVLs) are first sorted according to the value for <slot>. Then the code for initializing the constants is generated, followed by the code for initializing the variables.

The initialization is thus divided into the following phases:

1. The signatures are sorted according to the initialization slots. The slots are either defined implicitly or explicitly via the attribute 'global_init_slot'.
2. All constants are then initialized. This is done in the order of the slots. Signatures without constants are skipped.
3. Then the variables are initialized, again in the order of the slots.

Syntax: {attribute 'global_init_slot' := '<slot>'}

<slot>: Integer value that defines the position in the order of the calls. The default value for a POU (program, function block) is 50000. The default value for a GVL is 49990. The default value for static variables is 49980. A lower value results in earlier initialization.

Insertion location: The pragma always affects the entire GVL or POU and must therefore be above the VAR_GLOBAL declaration or the POU declaration.



If several programming blocks have been assigned the same value for the 'global_init_slot' attribute, the order of their initialization remains undefined. To avoid influencing the system behavior of TwinCAT 3, use values above 40000.

Sample:

The project contains two global variable lists GVL1 and GVL2. The MAIN program uses variables from both lists. For the initialization of a variable nA, GVL1 uses the variable nB, which is initialized in GVL2 with a value of 1000:

GVL1:

```
VAR_GLOBAL //49990
  nA : INT := GVL2.nB*3;
END_VAR
```

GVL2:

```
VAR_GLOBAL //49990
  nB : INT := 1000;
  nC : INT := 10;
END_VAR
```

MAIN program:

```
PROGRAM MAIN //50000
VAR
  nVar1 : INT := GVL1.nA;
  nVar2 : INT;
END_VAR
```

```
nVar1 := nVar + 1;
nVar2 := GVL2.nC;
```

In this case, the compiler issues an error because GVL2.nB is used to initialize GVL1.nA before GVL2 has been initialized. You can prevent this by setting the global_init_slot attribute to the position of GVL2 in the initialization sequence before GVL1.

To do this, the global_init_slot value of GVL2 must be smaller than the value of GVL1 (with the default value 49990) and greater than 40000 (reserved for system functions).

I.e.: 40001 <= global_init_slot value of GVL2 <= 49989.

GVL2:

```
{attribute 'global_init_slot' := '40500'}
VAR_GLOBAL
  nB : INT := 1000;
  nC : INT := 10;
END_VAR
```

The use of GVL2.nC in the implementation part of MAIN is uncritical even without using a pragma, since both GVLs are always initialized before the program.

16.8.2.16 Attribute 'hide'

The pragma prevents variables or program elements from being displayed in the TwinCAT interface. In online mode they are then not visible in the input assistant or in the declaration part, for example, they cannot be found using the cross-reference search, and no debugging functions (such as stepping or breakpoints) can be applied to them.

Note that variables declared with the 'hide' or 'hide_all_locals' attribute cannot be saved as persistent. Furthermore, the generation of the associated process image (allocated inputs/outputs) is prevented for "hidden" variables. Furthermore, no (ADS) symbols are generated for these variables. In other words, symbolic access is prevented.

The effect of 'hide' also affects variables and signatures within libraries that exist as .library.

Syntax: {attribute 'hide'}

Insertion location: Line above a signature, line above the declaration line of a variable (only the variable directly following the pragma is rendered invisible here) or, in the case of ST actions/transitions, directly at the start of the action/transition

Sample:

The function block FB_Sample uses the attribute {attribute 'hide'}:

```
FUNCTION_BLOCK FB_Sample
VAR_INPUT
    nA      : INT;
    {attribute 'hide'}
    bInvisible : BOOL;
    bVisible  : BOOL;
END_VAR
VAR_OUTPUT
    nB : INT;
END_VAR
```

In the main program an instance of the function block FB_Sample is defined.

```
PROGRAM MAIN
VAR
    fbSample : FB_Sample;
END_VAR
```

While an input value for fbSample is implemented, the function **List components**, which opens when typing "fbSample." (in the implementation part of MAIN) shows the variables nA, bVisible and nB, but not the hidden variable bInvisible.



The pragma [hide_all_locals](#) [▶ 806] can be used to hide all local variables of a declaration.



If the pragma "hide used in compiled libraries for variables and signatures" is used, these variables and signatures are not displayed in the library manager.

See also:

- [Attribute 'hide_all_locals'](#) [▶ 806]
- [Attribute 'conditionalshow'](#) [▶ 798]
- [Attribute 'conditionalshow_all_locals'](#) [▶ 799]

16.8.2.17 Attribute 'hide_all_locals'

This attribute works like the [Attribute 'hide'](#) [▶ 805]. The only difference is that the 'hide' attribute affects only one variable, where all local variables of a signature are affected by the 'hide_all_locals' attribute.

Note that variables declared with the 'hide' or 'hide_all_locals' attribute cannot be saved as persistent. Furthermore, the generation of the associated process image (allocated inputs/outputs) is prevented for "hidden" variables. Furthermore, no (ADS) symbols are generated for these variables. In other words, symbolic access is prevented.

Syntax: {attribute 'hide_all_locals'}

Insertion location: First line above the declaration part of the POU

Sample:

The function block FB_Sample uses the attribute:

```
{attribute 'hide_all_locals'}
FUNCTION_BLOCK FB_Sample
VAR_INPUT
```

```

    nA      : INT;
END_VAR
VAR_OUTPUT
    bB      : BOOL;
END_VAR
VAR
    nC, nD  : INT;
    bE      : BOOL;
END_VAR

```

The effect of the attribute 'hide' when using the attribute 'hide_all_locals' with the function block FB_Sample affects all local variables of the function block (nC, nD, and bE).

See also:

- [Attribute 'hide' \[► 805\]](#)

16.8.2.18 Attribute 'init_namespace'

The pragma causes a variable of type STRING or WSTRING, which is declared in a library function block with this pragma, to be initialized with the current namespace of the library when it is used in the project.

Syntax: {attribute 'init_namespace'}

Insertion location: Line above the declaration line of the variable in a library function block

Example:

The function block FB_Sample has the required attributes:

```

FUNCTION_BLOCK FB_Sample
VAR_OUTPUT
    {attribute 'init_namespace'}
    sNamespace : STRING;
END_VAR

```

An instance fbSample of the function block FB_Sample is defined within the main program MAIN

```

PROGRAM MAIN
VAR
    fbSample : FB_Sample;
    sMyNamespace : STRING;
END_VAR
sMyNamespace := fbSample.sNamespace;

```

The variable sNamespace is initialized with the current namespace, for example Tc3_TestLib. This value is assigned to sMyNamespace in the main program.

See also:

- [Library Manager \[► 276\]](#)

16.8.2.19 Attribute 'init_on_onlchange'

The pragma causes the variable, to which the pragma is applied, to be initialized with each online change.

● Fast online change

i For small changes (e.g. in the implementation section, with no shifting of variables required), a "fast online change" is performed. In this case, only the modified function block is compiled and reloaded. In particular, no initialization code is generated in this case. This also means that no code for initializing variables with the attribute 'init_on_onlchange' is generated. Usually this will not have any effect, since the attribute tends to be used to initialize variables with addresses, but a variable cannot change its address during a fast online change.

To ensure the init_on_onlchange attribute is applied to the entire application code, deactivate fast online change for the PLC project using the no_fast_online_change compiler definition. To this end, insert the definition in the [Compile category \[► 910\]](#) of the PLC project properties.

i The attribute 'init_on_onlchange' has no effect for individual FB variables

The Attribute 'init_on_onlchange' [▶ 807] only applies to global variables, program variables and local static variables of function blocks.

To reinitialize a function block during an online change, the function block instance must be declared with the attribute. The attribute is not evaluated for a single variable in a function block.

Syntax: {attribute 'init_on_onlchange' }

Insertion location: Line above the declaration line of a variable

16.8.2.20 Attribute 'initialize_on_call'

The pragma can be applied to input variables. It causes the input variables of a function block to be initialized each time the function block is called. If an input variable is affected that expects a pointer and this pointer was removed during an Online Change, the variable is initialized with zero.

Syntax: {attribute 'initialize_on_call'}

Insertion location: Always in the first line in the declaration part for the whole function block and additionally in a line above the declaration of the individual input variables

Sample:

If an input expects a pointer and this input is associated with the attribute, this pointer is reinitialized each time the function block is called. This can be used to prevent that a pointer is used that may have become invalid during an online change.

```
{attribute 'initialize_on_call'}
FUNCTION_BLOCK FB_Test
VAR_INPUT
    {attribute 'initialize_on_call'}
    pSetpoint : POINTER TO LREAL := 0;
END_VAR
VAR_OUTPUT
END_VAR
```

The pointer should be assigned when the function block is called.

```
fbTest(pSetpoint := ADR(fSetpoint));
```

16.8.2.21 Attribute 'instance-path'

The pragma can be applied to a local STRING variable and causes this local STRING variable to be initialized in sequence with the device tree path of the POU to which it belongs. This can be useful for error messages. The application of the pragma requires application of the attribute 'reflection' to the corresponding POU, as well as application of the additional attribute 'no_init' to the STRING variable itself.

Syntax: {attribute 'instance-path'}

Insertion location: Line above the line with the declaration of the STRING variable

Example:

The following function block contains the attributes 'reflection', 'instance-path' and 'noinit'.

```
{attribute 'reflection'}
FUNCTION_BLOCK FB_Sample
VAR
    {attribute 'instance-path'}
    {attribute 'noinit'}
    sPath : STRING;
END_VAR
```

An instance fbSample of the function block FB_Sample is defined within the MAIN program:

```
PROGRAM MAIN
VAR
    fbSample : FB_Sample;
    sMyPath : STRING;
```

```
END_VAR
fbSample();
sMyPath := fbSample.sPath;
```

Once the instance fbSample has been initialized, the string variable sPath is assigned the path of the instance fbSample, e.g. "<project>.MAIN.fbSample". In the main program this path is assigned to the variable sMyPath.



You can define the length of a string as required (exceeding 255 characters, if necessary). Note, however, that the string is truncated (from the rear) if it is assigned to a variable whose data type is smaller.

See also:

- [Attribute 'reflection' \[► 823\]](#)
- [Attribute 'noinit' \[► 815\]](#)

16.8.2.22 Attribute 'is_connected'

The 'is_connected' pragma is used to identify a Boolean function block variable that provides information on whether the assigned input of the function block receives an assignment when a function block instance is called.

The application of the pragma presupposes that the attribute 'reflection' is applied to the affected function block.

Syntax: {attribute 'is_connected' := '<input variable>'}

Insertion location: Line above the declaration of the individual Boolean function block variables

Sample:

In the function block FB a local variable is declared for each input variable (nIn1 and nIn2), and this is preceded by the attribute 'is connected' with specification of the input variable. The function block itself is assigned the pragma attribute 'reflection'.

When an instance of the function block is called, the local variable becomes TRUE if the input assigned to it has received an assignment.

```
{attribute 'reflection'}
FUNCTION_BLOCK FB_Sample
VAR_INPUT
    nIn1: INT;
    nIn2: INT;
END_VAR
VAR
    {attribute 'is_connected' := 'nIn1'}
    bIn1Connected: BOOL;
    {attribute 'is_connected' := 'nIn2'}
    bIn2Connected: BOOL;
END_VAR
```

Assumption: nIn1 receives an assignment from outside when the function block instance is called; nIn 2 receives no assignment at this time. The following code is created:

```
bIn1Connected := TRUE;
bIn2Connected := FALSE;
```

16.8.2.23 Attribute 'linkalways'

The pragma causes the associated object to be selected by the compiler and is thus always contained in the compiler information. This means that the object is always compiled and loaded on the PLC. The option **Always link** in the object properties, category **Advanced**, has the same effect.

Syntax: {attribute 'linkalways'}

Insertion location:

POU: First line above the declaration part of the POU

GVL: Line above the line with the keyword VAR_GLOBAL in the declaration part

Example:

Implementation of a POU (here: program) that uses the attribute 'linkalways':

```
{attribute 'linkalways'}
PROGRAM PRG_Test
  bSample : BOOL;
END_VAR
```

Implementation of a global variable list using the attribute 'linkalways':

```
{attribute 'linkalways'}
VAR_GLOBAL
  nVar1 : INT;
  nVar2 : INT;
END_VAR
```

16.8.2.24 Attribute 'minimal_input_size'

The pragma defines the minimum size for inputs on the stack. For C++-compatibility with C++ compilers for 32-bit-systems, each input takes up at least 4 bytes on the stack.

Syntax: {attribute 'minimal_input_size'}

Insertion location: Line above the method declaration

Sample:

Declaration of a PLC function block method that implements a C++-compatible interface:

```
{attribute 'c++_compatible'}
{attribute 'minimal_input_size' := '4'}
{attribute 'pack_mode' := '4'}
METHOD Method1 : HRESULT
VAR_INPUT
  nParameter1 : INT;
END_VAR
```

See also:

- [Attribute 'c++_compatible' \[► 795\]](#)

16.8.2.25 Attribute 'monitoring'

The pragma enables you to monitor values of properties or function calls in the online view of the IEC editor or in a watch list. There are two possible attribute values for this: 'variable' and 'call'.

Syntax:

```
{attribute 'monitoring' := 'variable'}
```

or

```
{attribute 'monitoring' := 'call'}
```

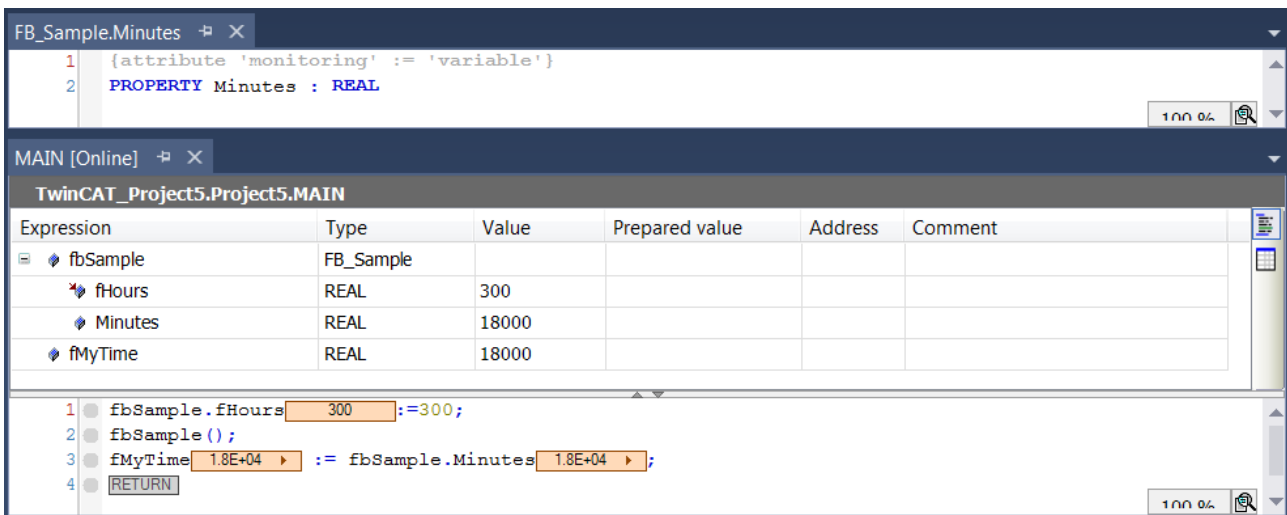
Monitoring of properties

You can monitor the subordinate properties in addition to the local variables in the online view of a function block or a program. You can thus monitor the values of the Get and Set methods.

Add either the {attribute 'monitoring' := 'variable'} or the {attribute 'monitoring' := 'call'} pragma in the declaration of the property. Then the current values of the property are automatically displayed in the IEC editor or in a watch list.

Sample:

TwinCAT displays the value for Minutes at the call position inline during online operation, because the declaration of the Minutes property contains the pragma {attribute 'monitoring' := 'variable'}.



For each application, check carefully which attribute pragma is suitable for displaying the required value. This depends on whether further operations involving the variable are implemented within the property.

Pragma 1 {attribute 'monitoring':='variable'}:

An implicit variable is created for the property, which is always assigned the current property value when the PLC program calls the Set or Get method. The last value stored in this variable is shown in the monitoring.

Pragma 2 {attribute 'monitoring':='call'}:

You can use this attribute only for properties that return simple data types or pointers, not for structured types. The value to be monitored is read or written by calling the property directly. This means that the monitoring service of the runtime system executes the Get or Set method of the property function.

i If you choose this monitoring type instead of using an implicit variable (as with {attribute monitoring':=' variable'}), you must consider possible side effects. Such side effects can occur when additional operations are implemented within the property function.

i The context menu command **Add to Watch** causes a variable, on which the cursor is currently positioned, to be added directly to a watch list in online mode.

i Forcing or writing of functions is not supported. You can, however, implement forcing implicitly by adding an additional input parameter for each function, which is used as internal force flag.

i Function monitoring is not possible in the compact runtime system.
 Note: Windows CE is not a compact runtime system. A compact runtime system does not support multitasking and usually has no file and operating system. A typical processor of a compact runtime system is for example an ARM Cortex™-M.

16.8.2.26 Attribute 'no_assign', Attribute 'no_assign_warning'

i Available from TwinCAT 3.1 Build 4026

The 'no_assign' pragma causes compiler errors to be issued when an instance of the function block is assigned to another instance of the same function block. Assignments like these should be avoided where possible if the function block contains pointers, as the pointers copied with the assignment lead to problems.

The 'no_assign_warning' pragma has the same effect as the 'no_assign' pragma but there is one difference: a compiler warning is issued instead of a compiler error.

Syntax: {attribute 'no_assign'} or {attribute 'no_assign_warning'}

Insertion location: First line in the declaration part of a function block

Sample:

As the FB_Test function block contains pointers, assignment of a function block instance should be avoided by adding the 'no_assign' attribute in the declaration of the FB_Test function block:

```
{attribute 'no_assign'}
FUNCTION_BLOCK FB_Test
VAR
    pVar      : POINTER TO LREAL;
...

```

Assignment of function block instances:

```
VAR_GLOBAL
    fbInst1 : FB_Test;
END_VAR

PROGRAM MAIN
VAR
    fbInst2 : FB_Test := fbInst1;
END_VAR

```

Then the following compiler error is output:

```
Assignment not allowed for type FB_Test
```

16.8.2.27 Attribute 'no_check'

The pragma prevents a check function (POUs for implicit checks) to be called for the POU. Since the check functions can affect the processing speed of the program, it may make sense to apply the attribute to function blocks, which have already been checked or are called regularly.

Syntax: {attribute 'no_check'}

Insertion location: First line in the declaration part of the POU

- [Object POU's for implicit checks](#) [► 163]

16.8.2.28 Attribute 'no_copy'

If an instance, for example a POU, is shifted in the memory during an Online Change, this necessitates a reallocation of this instance. The values of the variables contained in the instance are copied so that the variables have the same values after the Online Change as before it. If instances/variables have to be shifted during an Online Change, a dialog provides information about the effects and enables the Online Change to be aborted.

The effect of the pragma is that, in the course of the procedure to copy a shifted instance during the Online Change, no copying of the variable value of the variables contained in the instance takes place; instead, the variable is re-initialized in the course of the Online Change. This may make sense for a local pointer variable, which points to a variable that has just been moved by the online change and therefore has a modified address.

Syntax: {attribute 'no_copy'}

Insertion location: Line above the declaration line of the affected variable

16.8.2.29 Attribute 'no_explicit_call'

Add the pragma to a POU to prevent any direct call of this POU.

This is relevant, for example, for object-oriented function blocks, which can be controlled exclusively via methods (method function blocks) and properties (property function blocks). In this case, calls of the FB body would have no function or would not be allowed. If the body of a function block, which is declared with the attribute 'no_explicit_call', is nevertheless called directly, the compiler issues a compile error to indicate the incorrect use of the function block.

Syntax: {attribute 'no_explicit_call' := '<text value>'}

Insertion location: First line in the declaration part of a function block.

Example:

In the following example a function block is declared, whose FB body can be called directly ("FB_DirectCallAllowed"). A further function block is declared, whose FB body may not be called directly (FB_DirectCallNotAllowed). This function block is therefore declared with the attribute 'no_explicit_call'; the comment text: 'do not call this POU directly' is added to the attribute. In addition, an exemplary method is added to both function blocks.

Each of the two function blocks is instantiated once and called directly. The example methods are also called for the FB instances.

During compilation of this program, a compilation error is generated if the instance fbDirectCallNotAllowed is called directly (in this case: "do not call this POU directly"). The compiler does not block the three other calls.

Function block FB_DirectCallAllowed:

```
FUNCTION_BLOCK FB_DirectCallAllowed
VAR
END_VAR

METHOD SampleMethod
VAR_INPUT
END_VAR
```

Function block FB_DirectCallNotAllowed:

```
{attribute 'no_explicit_call' := 'do not call this POU directly'}
FUNCTION_BLOCK FB_DirectCallNotAllowed
VAR
END_VAR

METHOD SampleMethod
VAR_INPUT
END_VAR
```

Program MAIN:

```
PROGRAM MAIN
VAR
    fbDirectCallAllowed      : FB_DirectCallAllowed;
    fbDirectCallNotAllowed  : FB_DirectCallNotAllowed;
END_VAR

fbDirectCallAllowed();           // => OK
fbDirectCallAllowed.SampleMethod(); // => OK

fbDirectCallNotAllowed();       // => NOK: generates compile error "do not call this POU
directly"
fbDirectCallNotAllowed.SampleMethod(); // => OK
```

16.8.2.30 Attribute 'no_virtual_actions'

The pragma is applied to function blocks derived from a function block implemented in SFC and using the basic SFC sequence of this base class. The resulting actions show the same virtual behavior as methods. This means that the implementations of the actions in the base class from the derived class can be replaced by specific custom implementations.

If you apply the pragma to the base class, your actions are protected against overloading.

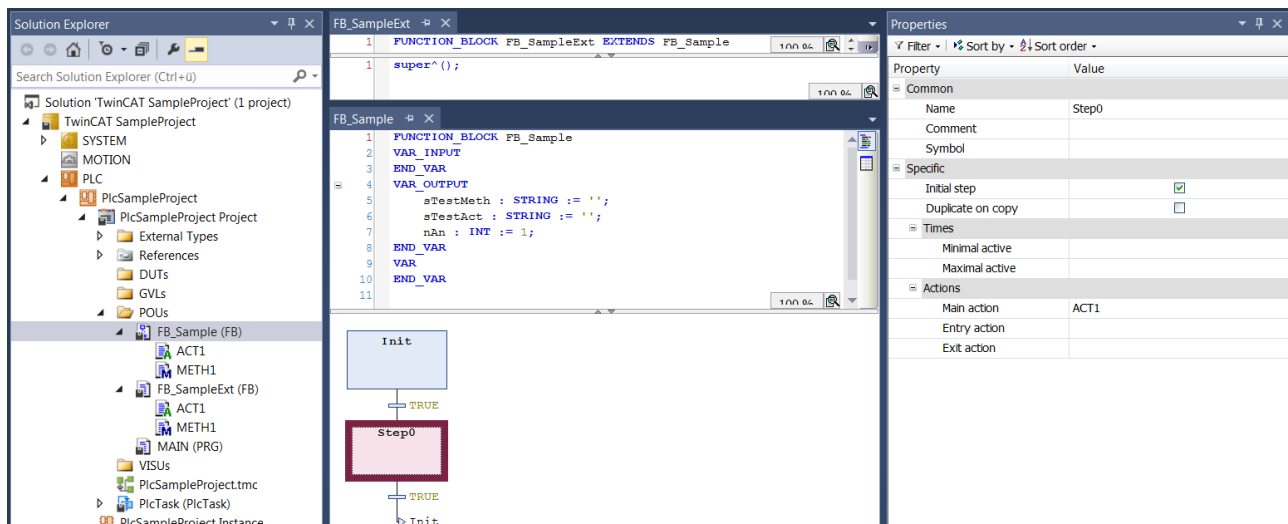
Syntax: {attribute 'no_virtual_actions'}

Insertion location: Top line in the declaration part of the function block

Sample:

The function block FB_Sample is the base class for the derived function block FB_SampleExt.

The derived class FB_SampleExt uses the special variable SUPER to call the base class process, which is written in SFC.



The sample implementation of this process is limited to the initial step followed by a single step with associated step action ACT1. This step with associated step action deals with assignment of the output variable.

```
nAn := nAn + 1; // Counting the action calls
sTestAct:='father_action';
METH1(); // Call of the method METH1 in order to set the string variable sTestMeth
```

In the case of the derived class `FB_SampleExt`, the step action is replaced by a special implementation of `ACT1`. `ACT1` differs from the original only by assigning the string 'child_action' instead of 'father_action' to the variable `sTestAct`.

In addition, the method `METH1`, which in the base class assigns the string 'father_method' to the variable `sTestMeth`, is overwritten so that `sTestMeth` is now assigned the value `sTestMeth`. The `MAIN` program calls an instance of the function block `FB_SampleExt` called "fbSampleExt". As expected, the value of the strings reflects the call of action and method of the derived class:

Expression	Type	Value
fbSampleExt	FB_SampleExt	
sTestMeth	STRING	'child_method'
sTestAct	STRING	'child_action'
nAn	INT	185

Now, however, you precede the base with the pragma `{attribute 'no_virtual_actions'}`:

```
{attribute 'no_virtual_actions'}
FUNCTION_BLOCK FB_Sample...
```

This changes the behavior: While the implementation of the derived class continues to be used for method `METH1`, calling the step action now results in a call of the action `ACT1` of the base class. Therefore, `sTestAct` is now assigned the value 'father_action':

Expression	Type	Value
fbSampleExt	FB_SampleExt	
sTestMeth	STRING	'child_method'
sTestAct	STRING	'father_action'
nAn	INT	177

16.8.2.31 Attribute 'no-exit'

The pragma suppresses the call of the FB_exit method of a function block for a certain of its instances.

Syntax: {attribute 'no-exit'}

Insertion location: Line before the declaration of the function block instance

Example:

The function block FB_Sample is added to the method FB_exit. Two instances of the function block FB_Sample are created in the MAIN program.

```
PROGRAM MAIN
VAR
    fbSample1 : FB_Sample;
    {attribute 'no-exit'}
    fbSample2 : FB_Sample;
END_VAR
```

fbSample1.FB_exit is called, fbSample2.FB_exit is not called.

See also:

- [Methods FB_init, FB_reinit and FB_exit \[► 848\]](#)

16.8.2.32 Attribute 'noflow' / 'flow'

The pragmas identify an area that is excluded from the representation of the flow control. See [Command Flow Control \[► 960\]](#).

Syntax:

```
{noflow}
```

```
{flow}
```

Insertion location: Rows above and below the area that should not be represented by the flow control.

Sample:

```
IF Test THEN
IF Test1 THEN
{noflow}
IF Test2 THEN
;
END_IF
{flow}
END_IF
END_IF
```

16.8.2.33 Attribute 'noinit'

The pragma is applied to variables that are not to be initialized implicitly.

Syntax:

```
{attribute 'no_init'}
```

```
{attribute 'no-init'}
```

```
{attribute 'noinit'}
```

Insertion location: Line above the declaration line of the affected variable

Example:

```
PROGRAM MAIN
VAR
  nA : INT;
  {attribute 'no_init'}
  nB : INT;
END_VAR
```

If the corresponding application is reset, the integer variable nA is once more implicitly initialized with 0, whereas the variable nB retains its current value.

16.8.2.34 Attribute 'obsolete'

The pragma causes a defined warning to be issued for a data type definition during compilation if the data type (structure, function block etc.) is used in the project. You can use this, for example, to indicate that a data type is no longer valid because an interface may have changed and this should be reflected in the project.

In contrast to a message pragma, this warning is defined centrally for all instances of a data type.

Syntax: {attribute 'obsolete' := 'user defined text'}

Insertion location: Line of the data type definition or one line above it.

Example:

The pragma is inserted in the definition of function block FB_Sample:

```
{attribute 'obsolete' := 'datatype FB_Sample() not valid!'}
FUNCTION_BLOCK FB_Sample
VAR_INPUT
  nVar : INT;
END_VAR
```

If you use FB_Sample as a data type, for example, in `fbSample : FB_Sample;`, a warning is issued when the project is compiled: "datatype FB_Sample not valid".

See also:

- [Message pragmas: \[► 793\]](#)

16.8.2.35 Attribute 'pack_mode'

The pragma defines how a data structure is packed during the allocation. The attribute must be inserted above the data structure and affects the packing of the whole structure.

Syntax: {attribute 'pack_mode' := '<pack mode value>'}

Possible values for <pack mode value>:

<pack mode value>	Associated package type	Description
0	aligned	All variables are assigned to byte addresses; no memory gaps occur.
1	1-byte aligned	
2	2-byte aligned	There are <ul style="list-style-type: none"> • 1-byte variables assigned to byte addresses • 2-byte variables assigned to addresses that are divisible by 2. The maximum gap that may occur is 1 byte • 4-byte variables assigned to addresses that are divisible by 2. The maximum gap that may occur is 1 byte • 8-byte variables assigned to addresses that are divisible by 2. The maximum gap that may occur is 1 byte • Strings always at byte addresses. No gap is created.
4	4-byte aligned	There are <ul style="list-style-type: none"> • 1-byte variables assigned to byte addresses • 2-byte variables assigned to addresses that are divisible by 2. The maximum gap that may occur is 1 byte • 4-byte variables assigned to addresses that are divisible by 4. The maximum gap that may occur is 3 byte • 8-byte variables assigned to addresses that are divisible by 4. The maximum gap that may occur is 3 byte • Strings always at byte addresses. No gap is created.
8	8-byte aligned	There are <ul style="list-style-type: none"> • 1-byte variables assigned to byte addresses • 2-byte variables assigned to addresses that are divisible by 2. The maximum gap that may occur is 1 byte • 4-byte variables assigned to addresses that are divisible by 4. The maximum gap that may occur is 3 byte • 8-byte variables assigned to addresses that are divisible by 8. The maximum gap that may occur is 7 byte • Strings always at byte addresses. No gap is created.

Insertion location: Line above the declaration of the data structure



Depending on the configuration of the structure, there may be no difference in the memory split between the individual modes. In this case the memory distribution of a structure with pack_mode = 4 can correspond to that of pack_mode = 8.



Arrays of structures: If the structures are grouped into arrays, bytes are inserted at the end of the structure so that the next structure is aligned again.

Example 1

```
{attribute 'pack_mode' := '1'}
TYPE ST_MyStruct:
STRUCT
    bEnable          : BOOL;
    nCounter         : INT;
    nMaxSize        : INT;
    bMaxSizeReached : BOOL;
    bReset          : BOOL;
END_STRUCT
END_TYPE
```

The memory area for a variable of data type ST_MyStruct is allocated "aligned: If the memory address of your component bEnable is 0x0100 for example, then the component nCounter follows at address 0x0101, nMaxSize at address 0x0103, bMaxSizeReached at address 0x0105 and bReset at address 0x0106. If 'pack_mode' := '2', then nCounter would be at 0x0102, nMaxSize at 0x0104, bMaxSizeReached at 0x0106 and bReset at 0x0107.

Example 2

```
STRUCT
  bVar1 : BOOL   := 16#01;
  nVar2 : BYTE   := 16#11;
  nVar3 : WORD   := 16#22;
  nVar4 : BYTE   := 16#44;
  nVar5 : DWORD  := 16#88776655;
  nVar6 : BYTE   := 16#99;
  nVar7 : BYTE   := 16#AA;
  nVar8 : DWORD  := 16#AA;
END_TYPE
```

	pack_mode = 0		pack_mode = 1		pack_mode = 2		pack_mode = 4		pack_mode = 8	
	Variable	Value	Variable	Value	Variable	Value	Variable	Value	Variable	Value
0	Var1	01	Var1	01	Var1	01	Var1	01	Var1	01
1	Var2	11	Var2	11	Var2	11	Var2	11	Var2	11
2	Var3	22	Var3	22	Var3	22	Var3	22	Var3	22
3	...	00	...	00	...	00	...	00	...	00
4	Var4	44	Var4	44	Var4	44	Var4	44	Var4	44
5	Var5	55	Var5	55						
6	...	66	...	66	Var5	55				
7	...	77	...	77	...	66				
8	...	88	...	88	...	77	Var5	55	Var5	55
9	Var6	99	Var6	99	...	88	...	66	...	66
10	Var7	AA	Var7	AA	Var6	99	...	77	...	77
11	Var8	AA	Var8	AA	Var7	AA	...	88	...	88
12	...	00	...	00	Var8	AA	Var6	99	Var6	99
13	...	00	...	00	...	00	Var7	AA	Var7	AA
14	...	00	...	00	...	00				
15					...	00				
16							Var8	AA	Var8	AA
17							...	00	...	00
18							...	00	...	00
19							...	00	...	00
20										
21										
22										
23										
24										
25										
26										
27										
28										
29										
30										
31										

Example 3

```

STRUCT
  nVar1 : BYTE := 16#01;
  nVar2 : LWORD := 16#11;
  nVar3 : BYTE := 16#22;
  nVar4 : BYTE := 16#44;
  nVar5 : DWORD := 16#88776655;
  nVar6 : BYTE := 16#99;
  nVar7 : BYTE := 16#AA;
  nVar8 : WORD := 16#AA;
END_TYPE
    
```

	pack_mode = 0		pack_mode = 1		pack_mode = 2		pack_mode = 4		pack_mode = 8	
	Variable	Value	Variable	Value	Variable	Value	Variable	Value	Variable	Value
0	Var1	01	Var1	01	Var1	01	Var1	01	Var1	01
1	Var2	11	Var2	11						
2	...	00	...	00	Var2	11				
3	...	00	...	00	...	00				
4	...	00	...	00	...	00	Var2	11		
5	...	00	...	00	...	00	...	00		
6	...	00	...	00	...	00	...	00		
7	...	00	...	00	...	00	...	00		
8	...	00	...	00	...	00	...	00	Var2	11
9	Var3	22	Var3	22	...	00	...	00	...	00
10	Var4	44	Var4	44	Var3	22	...	00	...	00
11	Var5	55	Var5	55	Var4	44	...	00	...	00
12	...	66	...	66	Var5	55	Var3	22	...	00
13	...	77	...	77	...	66	Var4	44	...	00
14	...	88	...	88	...	77			...	00
15	Var6	99	Var6	99	...	88			...	00
16	Var7	AA	Var7	AA	Var6	99	Var5	55	Var3	22
17	Var8	AA	Var8	AA	Var7	AA	...	66	Var4	44
18	...	00	...	00	Var8	AA	...	77		
19					...	00	...	88		
20							Var6	99	Var5	55
21							Var7	AA	...	66
22							Var8	AA	...	77
23							...	00	...	88
24									Var6	99
25									Var7	AA
26									Var8	AA
27									...	00
28										
29										
30										
31										

16.8.2.36 Attribute 'parameterstringof'

The pragma can be used to access the instance name of a variable using the visualization.

Syntax: {attribute 'parameterstringof' := '<variable>'}

Insertion location: Line above the declaration line of a variable

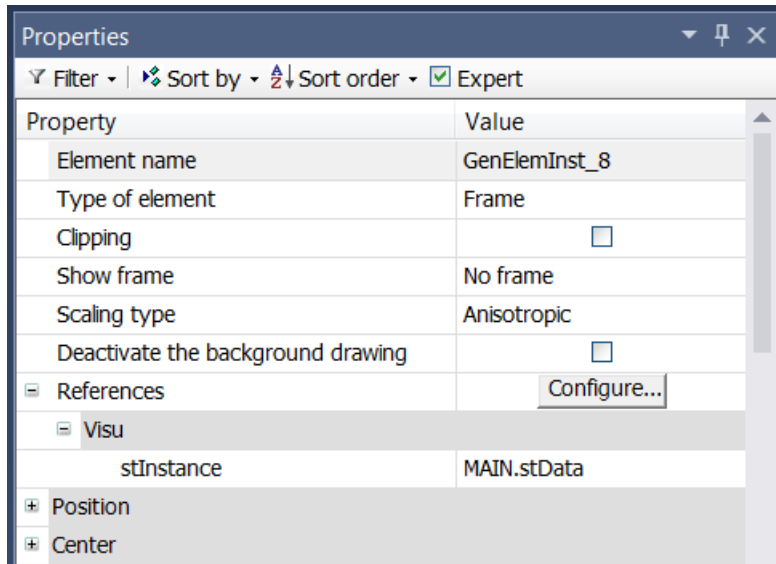
Example:

In the main program, the instance stData of the user-defined structure ST_Data was created:

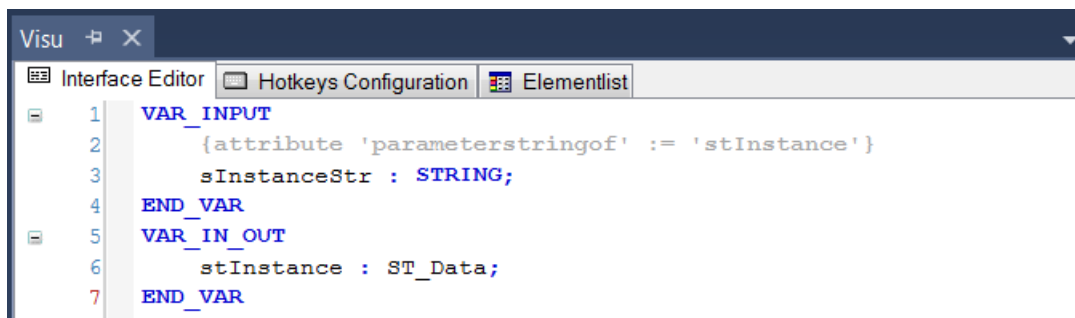
```
PROGRAM MAIN
VAR
  stData : ST_Data;
END_VAR
```

This instance is the input for a visualization "Visu" (in the input/output parameter stInstance). The visualization is referenced by a frame of another visualization, "MainVisu".

The settings of the frame element in "MainVisu" look like this:



The input/output variable stInstance and a further input variable sInstanceStr are declared in the interface editor associated with the visualization "Visu":



Although sInstanceStr is an input variable, it is not listed as input in the reference (see top image). This is because the variable sInstanceStr has the attribute 'parameterstringof' and is therefore automatically initialized with the name of the variable that was specified in the attribute. In the example, stInstance is the corresponding variable. The string variable sInstanceStr is therefore set to MAIN.stData and can now be used within the visualization "Visu", for example as a text variable for a placeholder "%s".

16.8.2.37 Attribute 'pin_presentation_order_inputs/outputs'

The pragmas are evaluated in the CFC and FBD/LD graphic editors and cause the inputs and outputs of the function block concerned to be displayed in the specified order. The order is programmed by assigning the names of the inputs/outputs to the attribute in the desired order.

Syntax:

```
{attribute 'pin_presentation_order_inputs' := '<First_Input_Name>, (<Next_Input_Name>)* ( *, )? (<Next_Input_Name>)* <Last_Input_Name>'}
```

```
{attribute 'pin_presentation_order_outputs' := '<First_Output_Name>, (<Next_Output_Name>)* ( *, )? (<Next_Output_Name>)* <Last_Output_Name>'}
```


- *
The optional terminal sign serves as a placeholder for all inputs/outputs that are not specified in the order of presentation. If the terminal sign is missing, the missing inputs/outputs will be appended at the end.
- (...)?
The content of the round bracket is optional.
- (...)*
The content of the round bracket is repeatedly optional and can therefore occur not at all, once or several times.

Insertion location: First line in the declaration part of a function block



Use of the attribute 'pin_presentation_order_inputs/outputs' in connection with the attribute 'pingroup'

This pragma is not evaluated if the pragma {attribute 'pingroup' := '<Group_Name>'} is used.

Sample:

1. Use of the attribute 'pin_presentation_order_inputs/outputs'

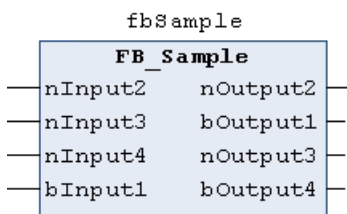
Function block FB_Sample:

```
{attribute 'pin_presentation_order_inputs' := 'nInput2,*,bInput1'}
{attribute 'pin_presentation_order_outputs' := 'nOutput2,nOutput1'}
FUNCTION_BLOCK FB_Sample
VAR_INPUT
    bInput1 : BOOL;
    nInput2 : INT;
    nInput3 : INT;
    nInput4 : INT;
END_VAR
VAR_OUTPUT
    bOutput1 : BOOL;
    nOutput2 : INT;
    nOutput3 : INT;
    bOutput4 : BOOL;
END_VAR
```

Program SampleProg:

```
PROGRAM SampleProg
VAR
    fbSample : FB_Sample;
END_VAR
```

In the presentation of the function block instance fbSample, the pragmas cause the following arrangement of the input and output pins:



See also:

- [Attribute 'pingroup' \[▶ 821\]](#)

16.8.2.38 Attribute 'pingroup'

The pragma causes the input or output pins (parameters) to be grouped together in the declaration of a function block. In the FBD/LD editor a pin group defined in this way can then be displayed as a unit in collapsed or expanded form at the added function block. Several groups are possible and are distinguished through their names. TwinCAT stores the corresponding state (collapsed) for each function block box with the project.

Syntax: {attribute 'pingroup' := '<group name>'}

Insertion location: Row above the declaration of the affected input or output variable in the declaration part of a function block.

Sample:

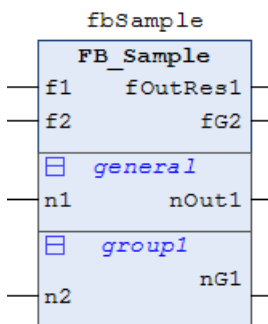
Function block FB_Sample:

```
FUNCTION_BLOCK FB_Sample
VAR_INPUT
    f1 : REAL;
    {attribute 'pingroup' := 'general'}
    n1 : INT;
    {attribute 'pingroup' := 'group1'}
    n2 : INT;
    f2 : REAL;
END_VAR
VAR_OUTPUT
    fOutRes1 : REAL;
    {attribute 'pingroup' := 'general'}
    nOut1 : INT;
    {attribute 'pingroup' := 'group1'}
    nG1 : INT;
    fG2 : REAL;
END_VAR
```

Program SampleProg:

```
PROGRAM SampleProg
VAR
    fbSample : FB_Sample;
END_VAR
```

In the presentation of the function block instance fbSample, the pragmas cause the following grouping of the input and output pins:



See also:

- [Attribute 'pin presentation order inputs/outputs' \[► 820\]](#)

16.8.2.39 Attribute 'qualified_only'

The pragma has the effect that variables from a global variable list can only be addressed by specifying the global variable list name (e.g. GVL.nVar). This also applies to variables of type enumeration, which can only be accessed by specifying enumeration name (e.g. E_Sample.eMember). This can be useful for avoiding confusion with local variables.

Please note that a GVL can only be assigned the attribute as a whole. Individual variable ranges of a GVL cannot be declared with the attribute.

Syntax: {attribute 'qualified_only'}

Insertion location: Line above the first VAR_GLOBAL in a GVL

Sample:

Global variable list (GVL):

```
{attribute 'qualified_only'}
VAR_GLOBAL
    nVar : INT;
END_VAR
```

Within a POU, for example MAIN, the global variable nVar can only be addressed using the prefix GVL:

```
GVL.nVar := 5;
```

The following incomplete call of the variables would generate an error:

```
nVar := 5;
```

16.8.2.40 Attribute 'reflection'

The pragma is used to indicate function blocks where TwinCAT should search for local STRING variables, to which the attribute 'instance-path' is applied. The compiler searches only function blocks marked with 'reflection' for variables with these attributes and thus needs less time.

Syntax: {attribute 'reflection'}

An example can be found in the description of the attribute 'instance-path'.

See also:

- [Attribute 'instance-path' \[► 808\]](#)

16.8.2.41 Attribute 'strict'

The attribute 'strict' results in compile errors occur in the following cases:

- Arithmetic operation with variables of the enumeration type
- Assignment of a constant value, which is not an enumeration value, to a variable of the enumeration type
- Assignment of a non-constant value, which has a data type other than the enumeration type, to a variable of the enumeration type

Syntax: {attribute 'strict'}

Insertion location: Line above the TYPE definition

16.8.2.42 Attribute 'subsequent'

The pragma is used to allocate variables directly one after the other at a memory location. If the list changes, the entire variable list is allocated to a new memory location. This pragma is used in programs and global variable lists.

Syntax: {attribute 'subsequent'}



VAR_TEMP in a program with attribute 'subsequent' causes a compiler error.



If a variable is in the 'RETAIN' list, the entire list is stored as 'RETAIN'.

16.8.2.43 Attribute 'Tc2GvlVarNames'

The pragma has the effect that symbols, which are declared in a GVL, are addressed via ADS just like in TwinCAT 2 (without the use of the GVL name as namespace).

Syntax: {attribute 'Tc2GvlVarNames'}

Example:

```
{attribute 'Tc2GvlVarNames'}
VAR_GLOBAL
    nVar AT %Q* : UINT;
END_VAR
```

16.8.2.44 Attribute 'TcCallAfterOutputUpdate'

The pragma defines whether a program is to be executed after an output update. This attribute replaces the TwinCAT 2 functionality of the option **IO at Task begin**.

Syntax: {attribute 'TcCallAfterOutputUpdate' }

Insertion location: This attribute must be added to all program POU's, which are to be called after the output update.

Example:

```
{attribute 'TcCallAfterOutputUpdate'}
PROGRAM MAIN
VAR
END_VAR
```

16.8.2.45 Attribute 'TcContextId'

Via the pragma, you can define which task should update an allocated variable.

Syntax: {attribute 'TcContextId' := '<TaskId>'}

The placeholder <TaskId> in inverted commas must be replaced by the task ID. The task IDs can be read in the **Result** table. The table is shown in the **Context** tab, which can be opened by double-clicking on the PLC process image (**<Project name> instance**).

Insertion location:

- Line above the first VAR_GLOBAL in a GVL
- Line above the declaration of a POU
- Line above the declaration line of the allocated variables

Sample 1:

Assumption: The task PlcTaskA has the ID 0, the task PlcTaskB has the ID 1.

The pragma is inserted in the line above the respective variable declaration. It therefore only affects the following declaration line.

The variable bVar1 is updated by the task PlcTaskA, the variable bVar2 is update by the task PlcTaskB.

```
VAR_GLOBAL
  {attribute 'TcContextId':='0'}
  bVar1 AT%Q* : BOOL;
  {attribute 'TcContextId':='1'}
  bVar2 AT%Q* : BOOL;
END_VAR
```

Sample 2:

Assumption: The task PlcTaskA has the ID 0, the task PlcTaskB has the ID 1.

The pragma is inserted in the GVL in the line above VAR_GLOBAL and in the line above a variable declaration.

Because the attribute is applied above its declaration line, the variable bVar3 is updated by the task PlcTaskB. Due to use above the first VAR_GLOBAL, all other variables (all up to bVar3) are updated by the task PlcTaskA.

```
{attribute 'TcContextId':='0'}
VAR_GLOBAL
  bVar1 AT%Q* : BOOL; // => PlcTaskA
  bVar2 AT%Q* : BOOL; // => PlcTaskA

  {attribute 'TcContextId':='1'}
  bVar3 AT%Q* : BOOL; // => PlcTaskB

  bVar4 AT%Q* : BOOL; // => PlcTaskA
END_VAR
```

Sample 3:

Assumption: The task PlcTaskA has the ID 0, the task PlcTaskB has the ID 1.

The pragma is inserted in the GVL in the line above the first VAR_GLOBAL as well as in the line above the second VAR_GLOBAL.

Please note that the purpose of this usage is not to have the variables bVar4-bVar6 updated by the task PlcTaskB.

Background: The insertion of the pragma above VAR_GLOBAL is only productive above the first VAR_GLOBAL in a GVL.

The sample illustrated leads to the following assignment: The pragma above the second VAR_GLOBAL is interpreted by the variable bVar4 as a pragma above the declaration line, therefore the variable bVar4 is updated by the task PlcTaskB. Due to use above the first VAR_GLOBAL, all other variables (all up to bVar4) are updated by the task PlcTaskA.

In order to apply the attribute to a whole group of variables (e.g. bVar4-bVar6), it is recommended to use a dedicated GVL for each group of variables.

```
{attribute 'TcContextId':='0'}
VAR_GLOBAL
bVar1 AT%Q* : BOOL; // => PlcTaskA
bVar2 AT%Q* : BOOL; // => PlcTaskA
bVar3 AT%Q* : BOOL; // => PlcTaskA
END_VAR
{attribute 'TcContextId':='1'}
VAR_GLOBAL
bVar4 AT%Q* : BOOL; // => PlcTaskB
bVar5 AT%Q* : BOOL; // => PlcTaskA
bVar6 AT%Q* : BOOL; // => PlcTaskA
END_VAR
```

16.8.2.46 Attribute 'TcContextName'

Via the pragma, you can define which task should update an allocated variable.

Syntax: {attribute 'TcContextName' := '<TaskName>'}

The placeholder <TaskId> in inverted commas must be replaced by the task name.

Insertion location:

- Line above the first VAR_GLOBAL in a GVL
- Line above the declaration of a POU
- Line above the declaration line of the allocated variables

Sample 1:

The pragma is inserted in the line above the respective variable declaration. It therefore only affects the following declaration line.

The variable bVar1 is updated by the task PlcTaskA, the variable bVar2 is update by the task PlcTaskB.

```
VAR_GLOBAL
  {attribute 'TcContextName':='PlcTaskA'}
  bVar1 AT%Q* : BOOL;
  {attribute 'TcContextName':='PlcTaskB'}
  bVar2 AT%Q* : BOOL;
END_VAR
```

Sample 2:

The pragma is inserted in the GVL in the line above VAR_GLOBAL and in the line above a variable declaration.

Because the attribute is applied above its declaration line, the variable bVar3 is updated by the task PlcTaskB. Due to use above the first VAR_GLOBAL, all other variables (all up to bVar3) are updated by the task PlcTaskA.

```
{attribute 'TcContextName':='PlcTaskA'}
VAR_GLOBAL
  bVar1 AT%Q* : BOOL; // => PlcTaskA
  bVar2 AT%Q* : BOOL; // => PlcTaskA

  {attribute 'TcContextName':='PlcTaskB'}
  bVar3 AT%Q* : BOOL; // => PlcTaskB
```

```

    bVar4 AT%Q* : BOOL; // => PlcTaskA
END_VAR

```

Sample 3:

The pragma is inserted in the GVL in the line above the first VAR_GLOBAL as well as in the line above the second VAR_GLOBAL.

Please note that the purpose of this usage is not to have the variables bVar4-bVar6 updated by the task PlcTaskB.

Background: The insertion of the pragma above VAR_GLOBAL is only productive above the first VAR_GLOBAL in a GVL.

The sample illustrated leads to the following assignment: The pragma above the second VAR_GLOBAL is interpreted by the variable bVar4 as a pragma above the declaration line, therefore the variable bVar4 is updated by the task PlcTaskB. Due to use above the first VAR_GLOBAL, all other variables (all up to bVar4) are updated by the task PlcTaskA.

In order to apply the attribute to a whole group of variables (e.g. bVar4-bVar6), it is recommended to use a dedicated GVL for each group of variables.

```

{attribute 'TcContextName' := 'PlcTaskA'}
VAR_GLOBAL
bVar1 AT%Q* : BOOL; // => PlcTaskA
bVar2 AT%Q* : BOOL; // => PlcTaskA
bVar3 AT%Q* : BOOL; // => PlcTaskA
END_VAR
{attribute 'TcContextName' := 'PlcTaskB'}
VAR_GLOBAL
bVar4 AT%Q* : BOOL; // => PlcTaskB
bVar5 AT%Q* : BOOL; // => PlcTaskA
bVar6 AT%Q* : BOOL; // => PlcTaskA
END_VAR

```

16.8.2.47 Attribute 'TcDisplayScale'

The pragma can be used to scale the value of a variable for the display.

Syntax: {attribute 'TcDisplayScale' := '<Value>'}

The following values are valid for <Value>:

- "+/-10"
- "0-10"
- "0-20"
- "4-20"
- "0-10(16)"
- "0-20(16)"
- "4-20(16)"
- "0.1°"
- "0.01°"
- "0-5"
- "0-30"
- "0-50"
- "+/-5"
- "+/-2.5"
- "+/-100"
- "0-5(16)"
- "0-30(16)"
- "0-50(16)"
- "+/-75mV"

Insertion location: Line above the declaration line of a variable

Example:

```
PROGRAM MAIN
VAR
  {attribute 'TcDisplayScale' := '0-10'}
  nVar AT %Q* : UINT;
END_VAR
```

16.8.2.48 Attribute 'TcEncoding'

Via the pragma you define how a variable of the type STRING should be interpreted.

Syntax: {attribute 'TcEncoding' := 'UTF-8'}

Insertion location: Line above the declaration line of the STRING variable

A variable of the type STRING declared in this way uses the UTF-8 format for the character formatting of the Unicode character set. As with every STRING variable the 0 termination is used here too.

Therefore, in order to support special characters and texts in different languages, the character set is not restricted to the typical character set of the data type STRING (or WSTRING). Instead, the Unicode character set in UTF-8 format is used in conjunction with the data type STRING. This format is generally used in IT.

If the ASCII character set is used, there is no difference between the typical formatting of a STRING and the UTF-8 formatting of a STRING.

With a STRING variable a single character is always stored inside one byte and with a WSTRING variable it is always stored inside two bytes. With the UTF-8 format, however, a single character is stored inside 1 - 4 bytes, depending on which character is concerned. For that reason, the number of characters often doesn't correspond to the size of the variable in bytes.

Sample 1:

The pragma is inserted in the line above the respective variable declaration. It therefore only affects the following declaration line.

```
VAR
  sMyStringText : STRING;
  wsMyWStringText : WSTRING;
  {attribute 'TcEncoding':='UTF-8'}
  sMyUTF8Text : STRING;
END_VAR
```

Sample 2:

The assignment of a literal requires a help function if the text contains characters that are not defined in the ASCII character set.

```
VAR
  sMyStringText : STRING := 'The dinner costs 30 €.';
  wsMyWStringText : WSTRING := "The dinner costs 30 €.";

  {attribute 'TcEncoding':='UTF-8'}
  sMyUTF8Text1 : STRING := sLiteral_TO_UTF8('The dinner costs 30 €.');
  {attribute 'TcEncoding':='UTF-8'}
  sMyUTF8Text2 : STRING := wsLiteral_TO_UTF8("The dinner costs 30 €.");
  {attribute 'TcEncoding':='UTF-8'}
  sMyUTF8Text3 : STRING := 'Hello World.';

  // verfügbar ab TC3.1 Build 4026
  {attribute 'TcEncoding':='UTF-8'}
  sMyUTF8Text4 : STRING := UTF8#'The dinner costs 30 €.';
END_VAR
```

Further help functions for STRING variables and conversion possibilities for UTF-8 are described in the [TwinCAT 3 PLC Lib TC2 Utilities documentation](#).

16.8.2.49 Attribute 'TcHideSubItems'

This pragma can be used to define which subelements of a variable are to be hidden. These are then no longer visible in the process image of the Solution Explorer.

Syntax: {attribute 'TcHideSubItems'}

Insertion location: Line above the declaration of the subelement

Example:

```
TYPE DUT :
STRUCT
a : INT;
{attribute 'TcHideSubItems'}
b : ARRAY [0..20000] OF BYTE;
END_STRUCT
END_TYPE
```

The attribute "TcHideSubItems" only creates the variable "b" and no longer also b[0], b[1], b[2]... b[19999] in the Solution Explorer.

16.8.2.50 Attribute 'TcIgnorePersistent'

The pragma can be used to prevent the recovery of a persistently stored value for a variable.

Syntax: {attribute 'TcIgnorePersistent' }

Insertion location: Line above the variable declaration

16.8.2.51 Attribute 'TcInitOnReset'

You can use the pragmas to define whether a persistent variable is to be reinitialized in case of a **Reset cold**.

Syntax: {attribute 'TcInitOnReset'}

Insertion location: Line above the declaration line of the affected variable



Available from TC3.1 Build 4024

Samples:

```
VAR PERSISTENT
nVar1 : UINT;
{attribute 'TcInitOnReset'}
nVar2 : UINT;
END_VAR
```

In case of a **Reset cold**, only nVar2 is reinitialized.

16.8.2.52 Attribute 'TcInitSymbol'

Using the pragma, you define whether a variable is used as an Init symbol. After the build process, the variables selected via this attribute are available in the tab "Symbol Initialization" of the PLC instance. In the "Value" column you can enter the desired values that are to be assigned to the variables before the start of the code execution. The values are copied into the variable value before the start of the code execution and overwrite the initial values that may have been specified during the variable declaration (e.g. nVar : INT := 123;).

From TC3.1 Build 4024.4, the initial value specified during the declaration is adopted into the table as initial value. See below for more information on this.

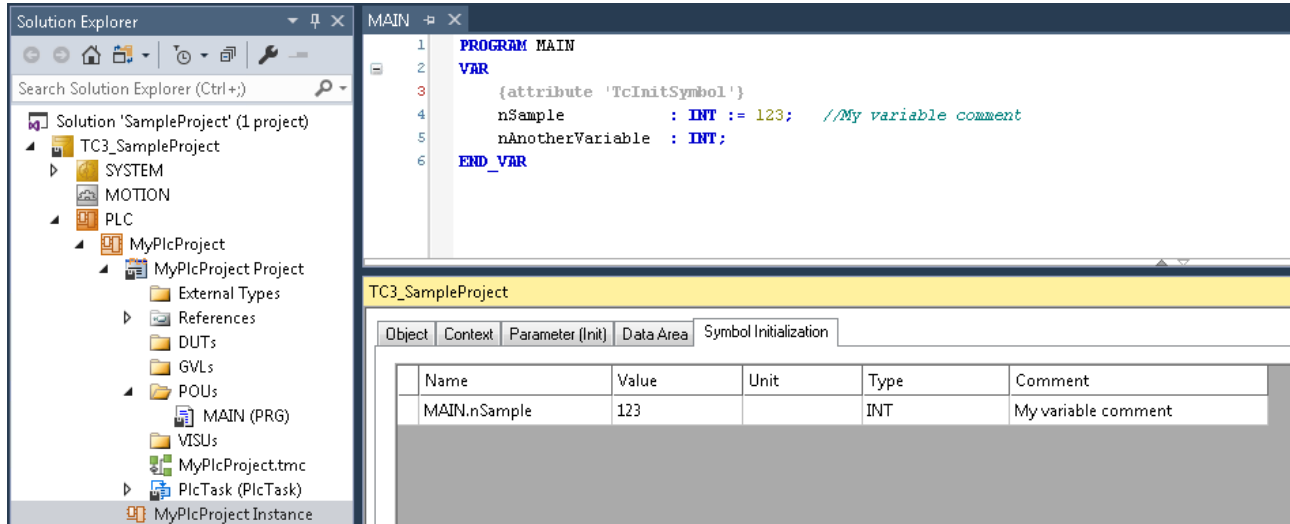
Syntax: {attribute 'TcInitSymbol'}

Insertion location: Line above the declaration line of the variables

Sample:

```
PROGRAM MAIN
VAR
  {attribute 'TcInitSymbol'}
  nSample      : INT := 123;  //My variable comment
  nAnotherVariable : INT;
END_VAR
```

This declaration leads to the following display on the **Symbol initialization** tab of the PLC instance:



Column filling:

The entries in the following columns are adopted from the variable declaration inside the PLC editor (refer also to the sample above):

- Name (= declaration path + symbol name)
- Value: Initial value of the variable declaration is only adopted into the table during the initial addition of the variables to the table; see below for more information.
- Type
- Comment

Initial entry in the table column "Value":

Scenario: A variable is declared with the attribute 'TcInitSymbol'. When this declaration is compiled for the first time, the variable is added to the symbol initialization table.

From TC3.1 Build 4024.4, the initial value of the variable declaration (123 in the above sample) is adopted into the table column "Value" during this addition of the variable to the table. With earlier TC3.1 versions, a zero initialization was used, so that the "Value" column was filled with the value 0.

Once the variable has been added to the symbol initialization table, you can enter the desired Init value in the table by changing the initial automatic entry. The variable nVar from the above sample is therefore initially added to the table with the value 123. If you subsequently change the table value to 456, the variable nVar is assigned the value 456 before the start of the code execution. The initial value of the variable declaration (123) is thus overwritten by the entry in the table (456).

Filling the table column "Unit":

The "Unit" column is used for OTCID data types and is automatically filled for these data types with the name of the TcCOM object, which belongs to the OTCID value in the "Value" column.

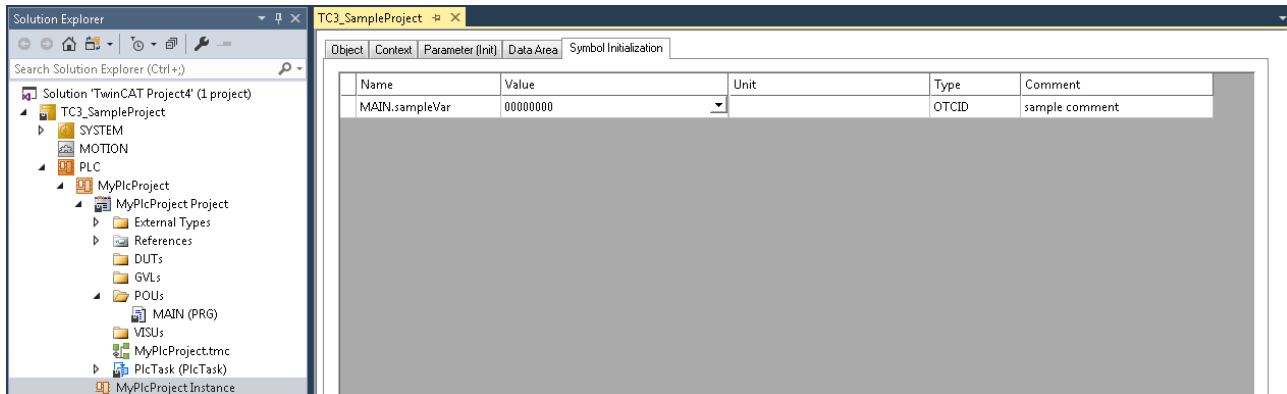
Sample:

```

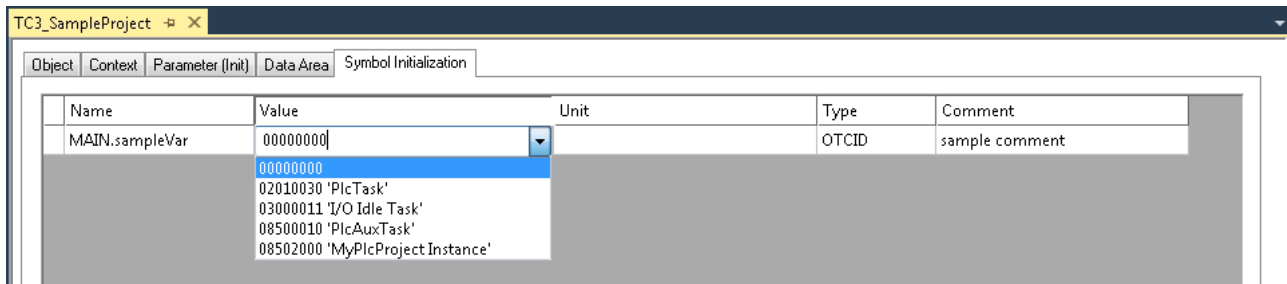
PROGRAM MAIN
VAR
  {attribute 'TcInitSymbol'}
  sampleVar      : OTCID;          //sample comment
END_VAR

```

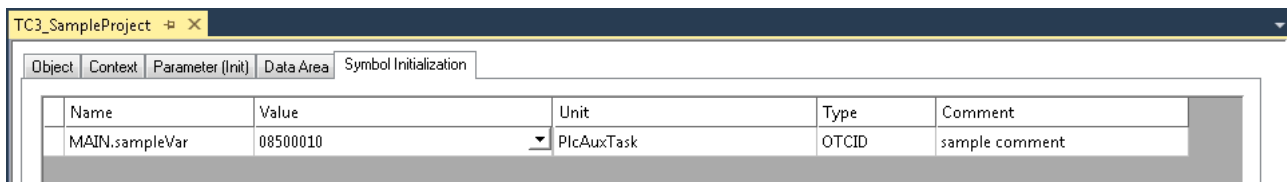
This declaration leads to the following display on the **Symbol initialization** tab of the PLC instance:



In order to provide a variable with a value, a selection menu can be opened that offers a choice of the possible values.



If in this case, for example, the value pair "08500010 'PlcAuxTask'" is selected, the ID (08500010) is entered in the "Value" column and the name (PlcAuxTask) is entered in the "Unit" column.



The value pair "Value" and "Unit" corresponds to the value pair that is shown on the **Parameter (Init)** tab.

Name	Value	CS	Unit	Type	PTCID
Project Name	MyPlcProject			STRING(12)	0x0850000C
Application Port	851			UINT	0x08500004
Auxtask Object Id	08500010		PlcAuxTask	OTCID	0x0850000B
Application Timestamp	2019-11-06T10:47:57			DT	0x0850000D
Symbolic Mapping	TRUE			BOOL	0x0850801B
Application Name	Port_851			STRING(8)	0x08500003
Task Module OIDs	[08502001]		MyPlcProject Instance##1	ARRAY [0..0] OF OTCID	0x08500005
Task Module SortOrders	[0]			ARRAY [0..0] OF UDINT	0x0850800F
Task Module Names	['PlcTask']			MSTRING(9)	0x08508019
Task Caller OIDs	[02010030]		PlcTask	ARRAY [0..0] OF OTCID	0x0850801A
Task Module ADIIndices	[0xffff, 0xffff]			ARRAY [0..1] OF WORD	0x0850801C

16.8.2.53 Attribute 'TcLinkTo' / 'TcLinkToOSO'

The two pragmas can be used to assign variables directly to inputs and outputs of another process images.

Syntax:

```
{attribute 'TcLinkTo' := '<I/O point name>'}
```

The placeholder <I/O point name> in inverted commas must be replaced by the name of the input or output.

```
{attribute 'TcLinkToOSO' := '<x,y,z>'I/O point name'}
```

x: Bit-offset of the PLC variables

y: Number bits to be connected

z: Offset of the target variable, on which these bits are to be mapped

The placeholder <I/O point name> in inverted commas must be replaced by the name of the input or output.

Linking of allocated variables of a function block:

Allocated variables of a function block can also be linked with the help of the pragma. For function blocks, this doesn't take place at the declaration of the allocated variables, but at the declaration point of the function block instance. Further information on this can be found below in the section "Examples of the use of the attribute".

Representation of the linking method:

If a variable is manually linked with an input or output, the associated variable in the process image is given a white icon, as is the linked channel. If on the other hand the variable is linked via the attribute 'TcLinkTo' / 'TcLinkToOSO', then the associated icon is blue. No icon exists with a variable that is not linked.

```
PROGRAM MAIN
VAR
  bVarIn1    AT%I*   : BOOL; // linked manually

  {attribute 'TcLinkTo' := 'TIID^Device 1 (EtherCAT)^Term 1 (EK1100)^Term 2 (EL1004)^Channel
2^Input'}
  bVarIn2    AT%I*   : BOOL; // linked via attribute

  bVarIn3    AT%I*   : BOOL; // not linked

  bVarOut1   AT%Q*   : BOOL; // linked manually

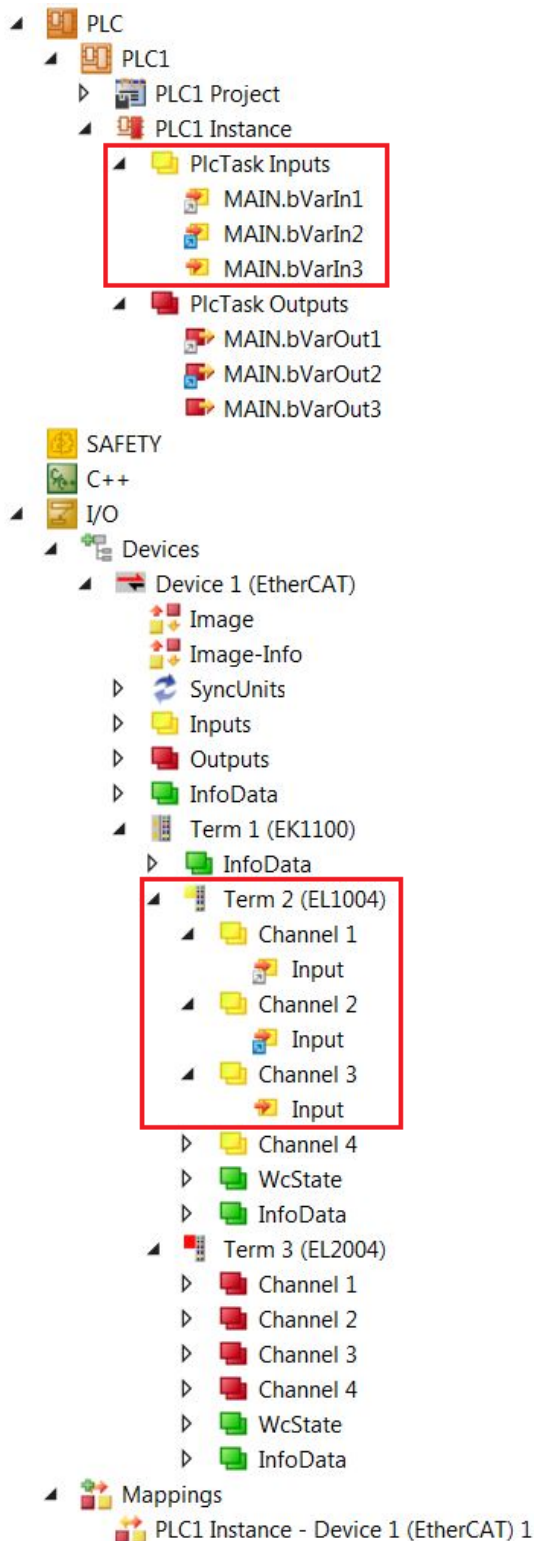
  {attribute 'TcLinkTo' := 'TIID^Device 1 (EtherCAT)^Term 1 (EK1100)^Term 3 (EL2004)^Channel
```

```

2^Output'})
  bVarOut2   AT%Q*   : BOOL; // linked via attribute

  bVarOut3   AT%Q*   : BOOL; // not linked
END_VAR

```



Examples of the use of the attribute:

```

PROGRAM MAIN
VAR
  nVar1           : INT;

  {attribute 'TcLinkTo' := 'TIID^Device 1 (EtherCAT)^Term 1 (EK1100)^Term 5 (EL4132)^Channel1^Output'}
  // Link when using the full path of an input or output

```

```

nVar2   AT%Q*   : INT;

(attribute 'TcLinkTo' := '[1] := TIID^Device 1 (EtherCAT)^Term 1 (EK1100)^Term 4 (EL2008)^Channel 1 4^Output;
[2] := TIID^Device 1
(EtherCAT)^Term 1 (EK1100)^Term 4 (EL2008)^Channel 5^Output'})
aVar3   AT%Q*   : ARRAY [1..2] OF BOOL;

(attribute 'TcLinkTo' := '.bIn := TIID^Device 1 (EtherCAT)^Term 1 (EK1100)^Term 3 (EL1008)^Channel 2^Input;
.bOut := TIID^Device 1 (EtherCAT)^Term 1 (EK1100)^Term 2 (EL2008)^Channel 2^Output')
// Link when using function blocks (FB_Module with allocated input bIn and allocated output bOut
)
fbVar4   : FB_Module;

(attribute 'TcLinkTo' := '[1].bIn := TIID^Device 1 (EtherCAT)^Term 1 (EK1100)^Term 2 (EL1008)^Channel 4^Input;
[1].bOut := TIID^Device 1 (EtherCAT)^Term 1 (EK1100)^Term 3 (EL2008)^Channel 4^Output;
[2].bIn := TIID^Device 1 (EtherCAT)^Term 1 (EK1100)^Term 2 (EL1008)^Channel 5^Input;
[2].bOut := TIID^Device 1 (EtherCAT)^Term 1 (EK1100)^Term 3 (EL2008)^Channel 5^Output')
aModule   : ARRAY[1..2] OF FB_Module;

(attribute 'TcLinkTo' := 'TIIB(2)^Channel 5^Output')
// Linking with the keyboard shortcuts
bVar5   AT%Q*   : BOOL;

(attribute 'TcLinkTo' := 'TIIB[TerminalXY]^Channel 5^Output')
// TerminalXY is the name of the terminal shown in the IO tree
nVar6   AT%Q*   : INT;

(attribute 'TcLinkToOSO' := '<1,2,3>TIIB(5)^Channel 2^Output')
// Link when using Offset/Size/Offset
nVar7   AT%Q*   : BYTE;

(attribute 'TcLinkToOSO' := '[1] := <1,2,3>TIIB(5)^Channel 2^Output;
[2] := <4,5,6>TIIB(5)^Channel 3^Output')
aVar8   AT%Q*   : ARRAY [1..2] OF WORD;

(attribute 'TcLinkTo' := '[0]
[0] := TIID^Device 1 (EtherCAT)^Term 1 (EK1100)^Term 2 (EL2008)^InfoData^State;
[1] := TIID^Device 1 (EtherCAT)^Term 1 (EK1100)^Term 3 (EL2008)^InfoData^State;
[2] := TIID^Device 1 (EtherCAT)^Term 1 (EK1100)^Term 4 (EL2008)^InfoData^State;
[0] := TIID^Device 1 (EtherCAT)^Term 1 (EK1100)^Term 5 (EL2008)^InfoData^State;
[1] := TIID^Device 1 (EtherCAT)^Term 1 (EK1100)^Term 6 (EL2008)^InfoData^State;
[2] := TIID^Device 1 (EtherCAT)^Term 1 (EK1100)^Term 7 (EL2008)^InfoData^State')
aVar9   AT%Q*   : ARRAY[0..1, 0..2] OF UINT;

(*****
Available from TC3.1 Build 4026
*****)

// ARRAY info on the left is "%d..%d" and on the right side "%d..#"
// Link info will be generate from 1 TO 4 (4 times) and from 3 TO 6 on channel side
(attribute 'TcLinkTo' := '[1..4] := TIID^Device 1 (EtherCAT)^Term 1 (EK1100)^Term 2 (EL2008)^Channel 3..#^Output')
bVar10 AT %Q* : ARRAY[1..4] OF BOOL;

(attribute 'TcLinkTo' := '[1..4] := TIID^Device 1 (EtherCAT)^Term 1 (EK1100)^Term 4 (EL4014)^AO Outputs Channel 1..#^Analog output')
bVar11 AT %Q* : ARRAY[1..4] OF INT;

// Also additional syntax for EtherCAT PLC structures to PLC mappings -
see "PLC" tab on EtherCAT terminals:
// ARRAY info on the left is "%d..%d" and on the right side "%d..#" -
channel index is added via "^%d" and the value is 0-based
(attribute 'TcLinkTo' := '[1..4] := TIID^Device 1 (EtherCAT)^Term 1 (EK1100)^Term 5 (EL2819)^^ 0..#')
bVar12 : ARRAY [1..16] OF MDP5001_280_700116F7;

END_VAR

```

Table of all available shortcuts:

Shortcut	To
TIIC	Node I/O configuration
TIID	node I/O configuration^I/O devices or node I/O configuration TAB "I/O devices"
TIRC	Node real-time configuration
TIRT	node real-time configuration^additional tasks or node "real-time configuration TAB "additional tasks"
TIRS	node real-time configuration^real-time settings or node "real-time configuration TAB "real-time settings"
TIPC	Node PLC configuration
TINC	Node NC configuration
TICC	Node CNC configuration
TIAC	Node CAM configuration
TING	Node axes
TINT	Node tables
TINS	Node SAF task
TIIT(%d)	Terminal %d (only for KL bus)
TIIB(%d)	Box %d
TIIF(%d)	Device %d
TIIT[%s]	Terminal with name %s (only for KL bus)
TIIB[%s]	Box with name %s
TIIF[%s]	Device with name %s

16.8.2.54 Attribute 'TcNcAxis'

The pragma can be used to associate NC axes outside the source code with variables of type Axis_Ref.

Syntax: {attribute 'TcNcAxis' := '<AxisName>'}

The placeholder <AxisName> in inverted commas must be replaced by the name of the axis, as described in the NC section.

Insertion location: Line above the declaration line of a variable

Examples:

```
PROGRAM MAIN
VAR
  {attribute 'TcNcAxis' := 'Axis1'}
  // Using the axis in the same POU of type Program.
  axis1      : AXIS_REF;

  {attribute 'TcNcAxis' := '.axis1:=Axis2;.axis2:=Axis3'}
  // Use of the axis in POU fbSample of type Function block.
  // The internal axis names (axis1 and axis2) of the POU fbSample must be assigned to the NC axes
  // used by the function block instance (axis 2 and axis 3).
  fbSample   : FB_Sample;

  {attribute 'TcNcAxis' := '[0]:=Axis4;[1]:=Axis5;[2]:=Axis6'}
  // Using an axis in an array.
  aAxis      : ARRAY [0..2] OF AXIS_REF;
END_VAR
```

16.8.2.55 Attribute 'TcNoSymbol' / 'tc_no_symbol'

The pragma can be used to specify that no (ADS) symbol is generated for a variable. In other words, symbolic access is prevented. If the value "unused" is assigned to the pragma, the pragma acts only for variables that are not used.

Note that variables declared with the attribute 'TcNoSymbol'/'tc_no_symbol' cannot be stored persistently. Furthermore, the generation of the associated process image (allocated inputs/outputs) is prevented for variables for which no (ADS) symbol is generated.

Syntax: {attribute 'tc_no_symbol'} or {attribute 'TcNoSymbol'}
(The notation 'TcNoSymbol' is only available with TwinCAT version 3.1.4022 or higher.)

Insertion location: Line above the declaration line of a variable

Samples:

```
VAR_GLOBAL
  {attribute 'tc_no_symbol'}
  nVar1 : INT;
  {attribute 'tc_no_symbol'}
  var2 : OTCID;
END_VAR
```

Or, for the new notation since TwinCAT version 3.1.4022:

```
VAR_GLOBAL
  {attribute 'TcNoSymbol'}
  nVar1 : INT;
  {attribute 'TcNoSymbol':='unused'}
  var2 : OTCID;
END_VAR
```

16.8.2.56 Attribute 'TcRpcEnable'

The pragma can be used to activate a method for a Remote Procedure Call (RPC). This allows the method to be called via ADS. The TwinCAT OPC UA server needs this pragma to make the method available as an OPC UA method.

Syntax: {attribute 'TcRpcEnable'}

Insertion location: First line above the declaration part of the method.

Sample:

```
{attribute 'TcRpcEnable'}
METHOD M_Sum : INT
VAR_INPUT
  nInput1 : INT;
  nInput2 : INT;
END_VAR
```

Further samples:

- ADS:
 - Link: TcAdsClient.InvokeRpcMethod Method (ITcAdsSymbol, Int32, .Object.)
 - Document path: TwinCAT 3 \ TE1000 XAE \ Technologies \ ADS \ TwinCAT ADS .NET \ TwinCAT.Ads Namespaces \ TwinCAT.Ads Namespace \ TcAdsClient Class \ TcAdsClient Methods \ TcAdsClient.InvokeRpcMethod Method \ TcAdsClient.InvokeRpcMethod Method (ITcAdsSymbol, Int32, .Object.)
- OPC UA:
 - Link: Method Call
 - Document path: TwinCAT 3 \ TFxxxx | TC3 Functions \ TF6xxx – Connectivity \ TF6100 TC3 OPC UA \ Technical introduction \ Server \ PLC \ Method call

16.8.2.57 Attribute 'TcRetain'

This pragma causes the following variables to be created as Retain Variables and to retain their value after an uncontrolled termination (power failure). The pragma thus represents an alternative to using the keyword `VAR RETAIN` [[▶ 691](#)].

The so-called Retain Handler ensures that the Retain Variables are written at the end of a PLC cycle and only in the corresponding area of the NovRam. The handling of the Retain Handler is described in chapter "[Retain data](#)" of the C/C++ documentation.

If self-defined Data Unit Types (DUTs) are to be used as Retain Variables, the data types must be available in the TwinCAT type system. You can either use the option **Convert to Global Type** or you can create structures directly as STRUCT RETAIN. However, the Retain Handler then handles all occurrences of the structure.

Retain Data cannot be used for POU's (function blocks) as a whole. However, individual elements of a POU can be used.

Syntax: {attribute 'TcRetain'}

Insertion location: Line above the declaration line of a variable

Sample:

```
PROGRAM MAIN
VAR
  {attribute 'TcRetain'}
  nVar1 : INT;
END_VAR
```

16.8.2.58 Attribute 'TcSwapDWord'

The pragma can be used to define that the word order should be changed for this allocated output variable (swapping of low word and high word). If the pragma is present, the checkbox "Swap LOWORD and HIWORD" is checked for the output within the PLC process image.

Syntax: {attribute 'TcSwapDWord'}

Insertion location: Line above the declaration line of an allocated output variable (AT%Q)

Examples:

```
VAR
  {attribute 'TcSwapWord'}
  nVar1 AT%Q* : WORD;
  {attribute 'TcSwapDWord'}
  nVar2 AT%Q* : DWORD;
END_VAR
```

16.8.2.59 Attribute 'TcSwapWord'

The pragma can be used to define that the byte order should be changed for this allocated output variable (swapping of low byte and high byte). If the pragma is present, the checkbox "Swap LOBYTE and HIBYTE" is checked for the output within the PLC process image.

Syntax: {attribute 'TcSwapWord'}

Insertion location: Line above the declaration line of an allocated output variable (AT%Q)

Examples:

```
VAR
  {attribute 'TcSwapWord'}
  nVar1 AT%Q* : WORD;
  {attribute 'TcSwapDWord'}
  nVar2 AT%Q* : DWORD;
END_VAR
```

16.8.2.60 Attribute 'to_string'

The pragma has an effect on how the result of the conversion of an enumeration component with the operator `TO STRING/TO WSTRING` [► 726] is output: If the enumeration declaration is provided with the pragma, the name of the enumeration component appears as a string instead of the numerical value.

Syntax: {attribute 'to_string'}

Insertion location: Line above the declaration of the enumeration.



Available from TC3.1 Build 4024

Sample:

Enumeration E_Sample

```
{attribute 'qualified_only'}
{attribute 'strict'}
{attribute 'to_string'}
TYPE E_Sample :
(
  eInit := 0,
  eStart,
  eStop
);
END_TYPE
```

Program MAIN

```
PROGRAM MAIN
VAR
  eSample      : E_Sample;
  nCurrentValue : INT;
  sCurrentValue : STRING;
  wsCurrentValue : WSTRING;

  sComponent   : STRING;
  wsComponent  : WSTRING;
END_VAR

nCurrentValue := eSample;
sCurrentValue := TO_STRING(eSample);
wsCurrentValue := TO_WSTRING(eSample);

sComponent := TO_STRING(E_Sample.eStart);
wsComponent := TO_WSTRING(E_Sample.eStop);
```

Result of the assignments/conversion functions:

- Value of nCurrentValue: 0
- Value of sCurrentValue: 'eInit'
- Value of wsCurrentValue: "eInit"
- Value of sComponent: 'eStart'
- Value of wsComponent: "eStop"

Result if the enumeration were not to be declared with the attribute 'to_string':

- Value of nCurrentValue: 0
- Value of sCurrentValue: '0'
- Value of wsCurrentValue: "0"
- Value of sComponent: '1'
- Value of wsComponent: "2"

See also:

- [Enumerations \[► 781\]](#)

16.8.3 Conditional pragmas

Conditional pragmas are used to influence the code generation in the precompile process or the compile process. The programming language ST supports these pragmas.



From TC 3.1 build 4024 you can use the conditional pragmas both in the declaration part and in the implementation part. Only the operators "defined (<identifier>)" and "hasvalue (<identifier>, '<value>')'" for simple compiler definitions are supported in the declaration part. The special uses of the two operators listed below are excluded from this. In addition, it is not possible to define the components of an enumeration or the data type of an alias with the help of conditional pragmas.

Conditional pragmas are supported in the PLC project, but not in libraries. In previous TC 3.1 versions the pragmas used in the declaration part were not evaluated.

With conditional pragmas you influence whether declaration or implementation code is taken into account for the compilation. You can make this dependent, for example, on whether a certain compiler definition is defined or has a certain value, whether a certain variable is declared, whether a certain function block exists, etc.

Pragma	Description
<p>Definition in the code:</p> <ul style="list-style-type: none"> • {define <identifier>} • {define <identifier> '<value>'} <p>Examples:</p> <ul style="list-style-type: none"> • {define variantA} • {define variantA '123'} • {define variantA 'true'} <p>Definition in the PLC project properties:</p> <ul style="list-style-type: none"> • <identifier> • <identifier> := '<value>' <p>Examples:</p> <ul style="list-style-type: none"> • variantA • variantA := '123' • variantA := 'true' • variantA := '123', variantB := '456' 	<p>The definition can be queried later with the defined operator.</p> <p>In addition, the optionally specified value can be queried and compared later with the hasvalue operator.</p>
{undefine <identifier>}	The {define} instruction of the <identifier> is canceled, and the identifier is then "undefined" again. If the specified identifier is currently not defined, the pragma is ignored.
{IF <expr>}... {ELSIF <expr>}... {ELSE}... END_IF}	<p>These are pragmas for conditional compilation.</p> <p>The specified expressions <expr> must be constant during the compilation; they are evaluated in the order in which they appear here, until one of the expressions shows a non-zero value. The text linked to the instruction is compiled; the other lines are ignored. The order of the sections is set. The ELSIF and ELSE sections are optional. The ELSIF sections may be used any number of times. Within the constant <expr> you can use multiple conditional compile operators.</p>
<expr>}	You can use one or several operators within the constant expression <expr> in a conditional compile pragma {IF} or {ELSIF}.

Different scopes

You can enter global define definitions as compiler definitions in the PLC project properties (Compile category) or at System Manager level one node above (Compiler Defines grouping, see documentation on variant management). The compiler definitions defined here are valid in the entire PLC project. Also, you can specify several define-definitions separated by commas in the PLC project properties.

Furthermore, you can define local compiler definitions in the declaration and implementation part of the PLC elements. These are then valid in the respective declaration or implementation editor.

See also:

- [Structured Text \(ST\), Extended Structured Text \(ExST\) \[► 123\]](#)
- Variant management

Operator defined (<identifier>)

The operator causes the expression to be assigned the value TRUE. A prerequisite is that the <identifier> was defined using a {define} instruction and was subsequently not undefined with an {undefine} instruction, in which case FALSE is returned.

Sample 1:

Requirement: There are two PLC projects, Plc1 and Plc2. Variable pdef1 is defined by a {define} instruction in Plc1, but not in Plc2.

```
{IF defined (pdef1)}
  (* This code is processed in Plc1 *)
  {info 'pdef1 defined'}
  hugo := hugo + SINT#1;
{ELSE}
  (* the following code is only processed in Plc2 *)
  {info 'pdef1 not defined'}
  hugo := hugo - SINT#1;
{END_IF}
```

In addition, a sample for a message pragma is included: Only the information pdef1 defined is shown in the message window when the PLC project is compiled, since pdef1 is actually defined. The message 'pdef1 not defined' is issued, if pdef1 is not defined.

Sample 2:

In the following sample, the variable sVariantUsed is initialized with different values depending on the valid compiler-define. In addition, either an allocated input variable or an output variable is declared. The variable nCounter is declared in any case, i.e. independent of the valid compiler definitions.

```
{IF defined (Variant1)}
  sVariantUsed      : STRING := 'Variant1';
  bOutput           AT%Q* : BOOL;
{ELSE}
  sVariantUsed      : STRING := 'NotVariant1';
  bInput            AT%I* : BOOL;
{END_IF}

nCounter            : INT;
```

Operators for the implementation part

The operators presented in the following can only be used in the implementation part and are not evaluated in the declaration part.

Operator defined (variable: <variable>)

This operator causes the expression to be assigned the value TRUE, if the variable <variable> is declared within the current scope; otherwise FALSE is returned.

Sample:

Requirement: There are two PLC projects, Plc1 and Plc2. Variable g_bTest is declared in Plc1, but not in Plc2.

```
{IF defined (variable: g_bTest)}
  (* the following code is only processed in Plc2*)
  g_bTest := x > 300;
{END_IF}
```

Operator defined (type: <identifier>)

This operator causes the expression to be assigned the value TRUE, if a data type with the identifier <identifier> is declared; otherwise FALSE is returned.

Sample:

Requirement: There are two PLC projects, Plc1 and Plc2. Data type DUT is declared in Plc1, but not in Plc2.

```
{IF defined (type: DUT)}
  (* the following code is only processed in Plc1*)
  bDutDefined := TRUE;
{END_IF}
```

Operator defined (pou: <pou name>)

This operator causes the expression to be assigned the value TRUE, if a function block with the name <pou name> is defined; otherwise FALSE is returned.

With the help of the syntax "pou: <pou name>.<method name>", you can also check whether the function block mentioned possesses methods or actions with the names mentioned.

Sample:

Requirement: There are two PLC projects, Plc1 and Plc2. CheckBounds function block exists in Plc1, but not in Plc2.

```
{IF defined (pou: CheckBounds)}
  (* the following code is only processed in Plc1 *)
  arrTest[CheckBounds(0,i,10)] := arrTest[CheckBounds(0,i,10)] + 1;
{ELSE}
  (* the following code is only processed in Plc2 *)
  arrTest[i] := arrTest[i]+1;
{END_IF}
```

Operator defined (IsLittleEndian)

The operator causes the expression to be set to FALSE, if the CPU is "BigEndian (Motorola Byte Order)".

Operator defined (IsFPUSupported)

If the expression returns TRUE, the code generator FPU (floating-point unit) generates code for calculations with REAL values. Otherwise TwinCAT emulates FPU operations, which is, however, significantly slower.

Operator hasvalue (RegisterSize, '<register size>')

<register size>: Size of a CPU register in bits

The operator causes the expression to return TRUE if the size of a CPU register equals <register size>.

Possible values for <register size>

- 16 for C16x,
- 64 for X86 64-bit
- 32 for X86

Operator hasvalue (PackMode, '<pack mode value>')

The checked PackMode depends on the device description, not on the pragma, which can be specified for individual DUTs.

Operator hasattribute (pou: <pou name>, '<attribute>')

Operator causes the expression to be assigned the value TRUE, if the attribute <attribute> is specified in the first line of the declaration part of the function block pou-name; otherwise FALSE is returned.

Sample:

Requirement: There are two PLC projects, Plc1 and Plc2. A function fun1 is defined in Plc1 and Plc2, but the attribute vision is also assigned to it in Plc1.

In Plc1:

```
{attribute 'vision'}
FUNCTION fun1 : INT
VAR_INPUT
  i : INT;
END_VAR
VAR
END_VAR
```

In Plc2:

```
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 Plc1 *)
ergvar := fun1(ivar);
{END_IF}
```

See also:

- [User-defined attributes \[► 794\]](#)

Operator hasattribute (variable: <variable>, '<attribute>')

Operator causes the expression to be assigned the value TRUE, if the variable is assigned the attribute '<attribute>' with the instruction {attribute '<attribute>'} in the line before the variable declaration; otherwise FALSE is returned.

Sample:

Requirement: There are two PLC projects, Plc1 and Plc2. The variable g_globalInt is used in Plc1 and Plc2, but the attribute 'DoCount' is also assigned to it in Plc1.

Declaration g_GlobalInt in Plc1

```
VAR_GLOBAL
  {attribute 'DoCount'}
  g_globalInt : INT;
  g_multiType : STRING;
END_VAR
```

Declaration g_GlobalInt in Plc2

```
VAR_GLOBAL
  g_globalInt : INT;
  g_multiType : STRING;
END_VAR
```

Pragma instruction:

```
{IF hasattribute (variable: g_globalInt, 'DoCount')}
(* the following code is only processed in Plc1 *)
  g_globalInt := g_globalInt + 1;
{END_IF}
```

See also:

- [User-defined attributes \[► 794\]](#)

Operator hastype (variable: <variable>, <type-spec>)

The operator causes the expression to return the value TRUE, if the variable <variable> is of data type <type-spec>; otherwise FALSE is returned.

Possible data types for <type-spec>:

```
BOOL | BYTE | DATE | DATE_AND_TIME | DINT | DWORD | INT | LDATE | LDATE_AND_TIME | LINT |
LREAL | LTIME | LTIME_OF_DAY | LWORD | REAL | SINT | STRING | TIME | TIME_OF_DAY | ULINT |
UDINT | UINT | USINT | WORD | WSTRING
```

Sample:

Requirement: There are two PLC projects, Plc1 and Plc2. The variable g_multitype is declared in Plc1 with the data type LREAL, but in Plc2 with the data type STRING.

```
{IF (hastype (variable: g_multitype, LREAL))}
  (* the following code is only processed in Plc1 *)
  g_multitype := (0.9 + g_multitype) * 1.1;
{ELSIF (hastype (variable: g_multitype, STRING))}
  (* the following code is only processed in Plc2 *)
  g_multitype := 'this is a multitalent';
{END_IF}
```

Operator hasvalue (<define-ident>, '<char-string>')

The operator causes the expression to return the value TRUE, if a variable with identifier <define-ident> is defined and has the value <char-string>; otherwise FALSE is returned.

Sample:

Requirement: There are two PLC projects, Plc1 and Plc2. The variable test is used in the PLC projects Plc1 and Plc2; in Plc1 it is given the value 1, in Plc2 the value 2.

The compiler definition can either be set in the PLC project properties via `test := '1'` or in the implementation part of the POU via `{define test '1'}`.

```
{IF hasvalue(test, '1')}
  (* the following code is only processed in Plc1 *)
  x := x + 1;
{ELSIF hasvalue(test, '2')}
  (* the following code is only processed in Plc2 *)
  x := x + 2;
{END_IF}
```

Operator NOT <operator>

The expression is assigned the value TRUE, if the inverse of <operator> returns the value TRUE. <operator> can be one of the operators described in this chapter.

Sample:

Requirement: There are two PLC projects, Plc1 and Plc2. PLC_PRG1 exists in Plc1 and Plc2; POU CheckBounds exists only in Plc1.

```
{IF defined (pou: PLC_PRG1) AND NOT (defined (pou: CheckBounds))}
  (* the following code is only processed in Plc2 *)
  bANDNotTest := TRUE;
{END_IF}
```

Operator <operator> AND <operator>

The expression is assigned the value TRUE if the two specified operators return TRUE. <operator> can be one of the operators described in this chapter.

Sample:

Requirement: There are two PLC projects, Plc1 and Plc2. PLC_PRG1 exists in Plc1 and Plc2; the POU CheckBounds exists only in Plc1.

```
{IF defined (pou: PLC_PRG1) AND (defined (pou: CheckBounds))}
  (* the following code is only processed in Plc1, *)
  bANDTest := TRUE;
{END_IF}
```

Operator <operator> OR <operator>

The expression returns TRUE, if one of the two operators returns TRUE. <operator> can be one of the operators described here.

Sample:

Requirement: There are two PLC projects, Plc1 and Plc2. PLC_PRG1 exists in Plc1 and Plc2; the POU CheckBounds exists only in Plc1.

```
{IF defined (pou: PLC_PRG1) OR (defined (pou: CheckBounds))}
  (* the following code is only processed in Plc1 and in Plc2 *)
  bORTest := TRUE;
{END_IF}
```

Operator (<operator>)

() enclose the operators.

See also:

- [Using pragmas \[▶ 137\]](#)
- [User-defined attributes \[▶ 794\]](#)

16.8.4 Region pragma

The pragma is used to consolidate multiple lines in a text editor to a block. The block can be assigned a name. Region pragmas can be nested.

Syntax: {region "description"} ... {endregion}

Be sure to follow this syntax so that the pragma is taken into account.

Code with region pragma: Extended and reduced view

The image shows two side-by-side code snippets. The left snippet shows the 'extended view' of code with a region pragma:


```
1
2 {region "description"}
3 //Code
4 //Code
5 {endregion}
6
7 //Code
8
```

 The right snippet shows the 'reduced view' where the region pragma is collapsed into a single line:


```
1
2 {region "description"} [3 lines]
6
7 //Code
8
```

 The line numbers 1, 2, 6, 7, and 8 are visible in both snippets, indicating the mapping between the two views.

The pragma can be used in the ST editor and all declaration editors. Syntax highlighting can be adapted in the options.

16.8.5 Pragmas for warning suppression

16.8.5.1 Pragmas 'warning disable', 'warning restore'

The pragma {warning disable <compiler ID>} causes certain warnings to be suppressed.

The pragma {warning restore <compiler ID>} reactivates a suppressed message.

Syntax:

```
{warning disable <compiler ID>}
```

```
{warning restore <compiler ID>}
```

<compiler ID>: ID, which is located at the start of a warning message or in the overview of the compiler warnings in the PLC project properties.

Sample:

Compiler warning:

```
C0195: Implicit conversion from unsigned Type 'UINT' to signed Type 'INT' : possible change of sign
```

Applying the pragma to a variable declaration:

```
VAR
  {warning disable C0195}
  test1 : UINT := -1;
  {warning restore C0195}
  test2 : UINT := -1;
END_VAR
```

test1 does not generate an error message, test2 generates an error message.

● Pragmas in the implementation editor

i If you wish to use the pragmas to suppress warnings in the implementation editor, this is currently possible in the ST editor as well as in the FBD/LD/IL editor.

In FBD/LD/IL the desired pragma must be entered in a [label](#) [▶ 661].

See also:

- TC3 User Interface documentation: Command Properties (PLC project) > [Category Compiler Warnings](#) [▶ 919]

16.8.5.2 Attribute 'suppress_wrn_C0410'

The attribute 'suppress_wrn_C0410' must be used instead of the pragma 'warning disable' for the compiler warning with the ID C0410.

Syntax: {attribute 'suppress_wrn_C0410'}

Example:

Compiler warning for a property:

```
C0410: COMPATIBILITY WARNING: A write access to a Property of type REFERENCE calls the SET-Accessor for versions < 3.1.4022.0 and writes the reference, but calls the GET-Accessor for versions >= 3.1.4022.0 and writes the value! Use the operator REF= if you want to assign the reference.
```

Applying the attribute to a property:

```
{attribute 'suppress_wrn_C0410'}
PROPERTY refSample : REFERENCE TO BOOL
```

On account of the attribute the warning C0410 will not be generated for the property refSample.

16.9 Identifier

i Note also the TwinCAT 3 programming conventions (section "TwinCAT 3 programming conventions")

Rules for identifier allocation

Rules for the identifier of a variable

- An identifier must contain the following letters and numbers:
 - 0...9
 - A...Z
 - a...z
- TwinCAT is not case-sensitive, that is to say: VAR1 and var1 denote the same variable.
- An identifier may not begin with a number.
- An identifier may not contain spaces, special characters (!, &, ß,...) or hyphens.

- However, the use of underscores is allowed and TwinCAT recognizes these. That means that A_BCD and AB_CD, for example, are treated as two different identifiers. Usually no separators, such as the underscore, are used between the words.
- Double underscores ('__') should be avoided, because they are used for internal variables.
- There is no limit to the length of an identifier.

Rules for the multiple use of identifiers (namespaces)

- A local identifier may only be used once.
- An identifier must not be identical to a keyword (for example variable type).
- A global identifier may be used repeatedly. If a local variable has the same name as a global variable, the local variable has priority within a POU.
- A variable that is defined in a global variable list can have the same name as a variable that is defined in another GVL. TwinCAT offers the following standard-extending functionalities with regard to the namespace/scope of variables:
 - Operator - Global namespace: An instance path that starts with a dot always opens a global namespace. If there is a local variable, for example nVar, which has the same name as a global variable, .nVar is used to address the global variable.
 - The name of a global variable list can unambiguously define the namespace for the included variables. In this way, you can declare variables with the same name in different global variable lists and still address them uniquely by prefixing the list name.
Example:
`GVL1.nvar := GVL2.nVar; (*nVar from GVL2 is copied to nVar in GVL1*)`
 - Variables that are defined in the global variable list of a library that is integrated in the project can be addressed unambiguously according to the following syntax: <namespace library>.<name of the GVL>.<variable name>.
Example:
`GVL1.nVar := Lib1.GVL1.nVar (* nVar from GVL1 in library Lib1 is copied of nVar in GVL1*)`
- For a library, a namespace is defined when it is inserted via the library manager. In this way, you can unambiguously address a library function block or a library variable via <namespace library>.<function block name|variable name>. For nested libraries, note that the namespaces of all participating libraries have to be specified in sequence.
Example: If Lib1 is referenced by Lib0, the function F_Sample contained in Lib1 is addressed via
`Lib0.Lib1.F_Sample:nVar := Lib0.Lib1.F_Sample(nValue:=4); (*Return value of F_Sample is copied to variable nVar in the project*)`

See also:

- [Variables \[▶ 682\]](#)
- [Data types \[▶ 756\]](#)

16.10 Shading rules

In TwinCAT it is in principle allowed to use the same identifier for different elements. For example, a function block and a variable can be named the same. However, to prevent confusion, this should be avoided.

Negative example:

In the following code snippet, a local function block instance has the same name as a function:

```
FUNCTION Sample : INT
FUNCTION_BLOCK FB_Sample
PROGRAM MAIN
VAR
    Sample : FB_Sample;
END_VAR
Sample();
```

In such a case it is unclear whether the instance or the function is called in the program.

Naming conventions:

To ensure that names are always unique, naming conventions should be followed, for example for specifying certain prefixes for variables. For a possible definition of such prefixes, see the Identifier/name section of TwinCAT 3 programming conventions.

Naming conventions can be checked automatically using TE1200 | PLC Static Analysis. Static code analysis could also detect duplicate use of the name `Sample` by checking rule SA0027 and report it as an error.

Qualified access:

Ambiguous situations can also be avoided by consistently using the 'qualified_only' [▶ 822] attribute for enumerations and global variable lists and by using qualified libraries.

Shading:

The compiler basically reports neither errors nor warnings when the same identifier is used for different elements. Instead, the compiler searches the code in a specific order for the declaration of the identifier. If a declaration was found, then the compiler does not search for possible further declarations elsewhere. If further declarations exist, then these are "shadowed" for the compiler. The following describes the shadowing rules, this means, the search order that the compiler uses when searching for the declaration for identifiers. In the section "Ambiguous accesses and qualified accesses", ways are shown to avoid ambiguous accesses and to circumvent the shadowing rules.

Search order in the application

When the compiler encounters a single identifier in the code of an application, it looks for the associated declaration in the following order:

1. Local variables of a method
2. Local variables in the function block, program or function and in any basic function blocks
3. Local methods of the function block
4. Global variables in the project if the 'qualified_only' attribute is not set in the variable list where the global variables are declared.
5. Global variables in attracted libraries when neither the library nor the variable list requires qualified access.
6. Function block or type names from the project (i.e.: names of global variable lists, function blocks, etc.)
7. Function block or type names from a library
8. Namespaces of locally attracted libraries and libraries published by libraries

Search order in the library

When the compiler encounters a single identifier in the code of a library, it looks for the associated declaration in the following order:

1. Local variables of a method
2. Local variables in the function block, program or function and in any basic function blocks
3. Local methods of the function block
4. Global variables in the local library if the variable list in which the global variables are declared does not have the 'qualified_only' attribute set.
5. Global variables in attracted libraries when neither the library nor the variable list requires qualified access.
6. Function block or type names from the local library (e.g. names of global variable lists, function blocks, etc.)
7. Function block or type names from an attracted library
8. Namespaces of locally attracted libraries and libraries published by locally attracted libraries.

Ambiguous accesses and qualified accesses

Despite these search orders, ambiguous accesses may occur. This is the case, for example, when a variable with the same name occurs in two global variable lists that require unqualified access. Such a case is reported by the compiler as an error (for example: ambiguous use of the name <variable>).

Such an ambiguous use can be made unambiguous by a qualified access, for example by accessing it via the name of the global variable list (for example: `GVL.<Variable>`).

Qualified access can also always be used to bypass shading rules.

- The name of the global variable list can be used to uniquely access a variable in this list.
- The name of a library can be used to uniquely access elements in that library.
- The pointer [THIS \[► 694\]](#) can be used to uniquely access variables in a function block, even if a local variable with the same name exists in a method of the function block.

To find the declaration point of an identifier at any time, select the [Command Go To Definition \[► 880\]](#), available in the context menu of the editor window. This can be especially helpful if the compiler produces a seemingly incomprehensible error message.

Search in instance paths

The search orders described above do not apply to identifiers that appear in an instance path as a component, or to identifiers that are used as inputs in calls.

For an access of the type `yy.Komponente` it depends on the entity described by `yy` where to look for the declaration of `Komponente`.

- If `yy` refers to a variable with structured data type (i.e. of type `STRUCT` or `UNION`), then `Komponente` is searched in that order:
 - Local variables of the function block
 - Local variables of the basic function block
 - Methods of the function block
 - Methods of the basic function block
- If `yy` refers to a global variable list or a program, then `Komponente` is searched only in this list.
- If `yy` refers to a namespace of a library, then `Komponente` is searched in this library in the same way as described in the above section "Search order in the library".

Only in the second instance does the compiler decide whether access to the element found is permitted, i.e. whether the variable is possibly only accessible locally, or whether a method is private. If access is not allowed, an error is issued.

See also:

- [Identifier \[► 844\]](#)
- [Attribute 'qualified_only' \[► 822\]](#)
- [THIS \[► 694\]](#)

16.11 Keywords

Keywords that identify scopes, data types or operators must be written in capitals in all editors.

Keywords cannot be used as variable names.

Samples:

ABS, ACOS, 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_CASE, END_FOR, END_IF, 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_CONFIG, VAR_EXTERNAL, VAR_GLOBAL, VAR_IN_OUT, VAR_INPUT, VAR_OUTPUT, VAR_STAT, VAR_TEMP

WHILE, WORD, WSTRING

XOR, XORN

TwinCAT automatically checks the correct use of keywords and identifies errors with a red squiggly line immediately when they are entered.



When TwinCAT creates implicit code, variables and functions are usually given a name that contains "_". The use of double underscores in the implementation code is automatically prevented. Thus, there can be no conflicts between the system internal and the identifiers assigned by the programmer.

The following keywords are used in the TwinCAT export format. Therefore, you must not use them as identifiers:

- ACTION
- END_ACTION
- END_FUNCTION
- END_FUNCTION_BLOCK
- END_PROGRAM

16.12 Methods FB_init, FB_reinit and FB_exit

You can use the methods explicitly to influence the initialization of function block variables and the behavior when function blocks are terminated.

- [FB_init \[► 850\]](#)
- [FB_reinit \[► 853\]](#)
- [FB_exit \[► 854\]](#)

Refer also to the information on the different [operating cases](#) [► 855] and the [behavior of these methods with derived function blocks](#) [► 859].

i The type of return value for the implicit methods is BOOL. The return type should not be changed even if the value is not evaluated.

Do not add any output variables to the methods. You can only define additional input variables.

i Explicit calling is not recommended

The methods FB_init, FB_reinit and FB_exit are system functions that are called implicitly at different times (further information on this can be found under [Operating cases](#) [► 855]). Explicitly calling these methods can have inadvertent consequences and is therefore not recommended.

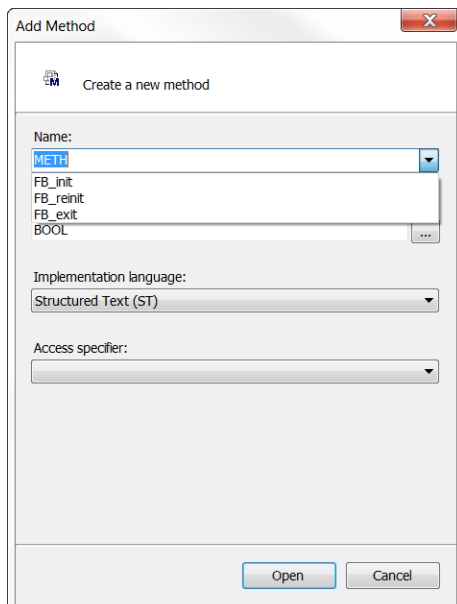
i Automatic core dump on exception in FB_init/FB_reinit/FB_exit

If an exception error occurs within the FB_init/FB_reinit/FB_exit code, e.g. due to a programming error, the runtime system automatically stores a core dump on the target system (from TC3.1 Build 4024.25). This core dump is stored as a *.core file in the boot folder of the target system (by default under C:\TwinCAT\3.1\Boot\Pic) and can be used for the search of causes.

For more information on loading a core dump, see: [Error analysis with core dump](#) [► 247]

Add methods

1. Select a function block in the PLC project tree in the **Solution Explorer**.
2. In the context menu select the command **Add > Method...**
 - ⇒ The dialog **Add Method** opens.
3. Open the drop-down list in the field for naming the method.
4. Select one of the methods FB_init, FB_reinit or FB_exit.
5. Click on **Open**.
 - ⇒ The method is added to the PLC project tree and opened in the editor. The definition of the method (return value, parameter) takes place automatically.



Return value

Implicit calls

If the methods are called implicitly the return value will not be evaluated by the system. Even if you adjust the return value, it will not be evaluated with an implicit call.

Explicit calls

If the methods are called explicitly you can evaluate the return value. You can thus return a meaningful return value.

See also:

- [Object Method](#) [► 90]
- [Attribute 'call_after_init'](#) [► 796]
- [Attribute 'call_after_online_change_slot'](#) [► 797]
- [Attribute 'call_on_type_change'](#) [► 797]
- [Attribute 'no_copy'](#) [► 812]
- [Attribute 'no-exit'](#) [► 815]

16.12.1 FB_init

FB_init is always implicitly available and is generally called in order to perform the standard initialization (implicit call). You can also explicitly declare the method and add additional code to the standard initialization code to exert specific influence.

If the explicitly defined initialization code is reached during the execution, the function block instance is already fully initialized via the implicit initialization code in the case of both automatic zero initialization and individual value initialization. Therefore, no `SUPER^.FB_init` call is allowed.

● Debugging

i Finding errors in the FB_init methods is laborious because, among other things, setting breakpoints cannot have the desired effect.

● Automatic core dump on exception in FB_init/FB_reinit/FB_exit

i If an exception error occurs within the FB_init/FB_reinit/FB_exit code, e.g. due to a programming error, the runtime system automatically stores a core dump on the target system (from TC3.1 Build 4024.25). This core dump is stored as a *.core file in the boot folder of the target system (by default under `C:\TwinCAT\3.1\Boot\Pic`) and can be used for the search of causes.

For more information on loading a core dump, see: [Error analysis with core dump](#) [► 247]

● Observe the order of initialization

i Please note that the values assigned to the input variables of the function block instance are not yet known when FB_init is executed. If you wish to parameterize the execution of FB_init, you can declare additional method inputs in the FB_init method.

For more information see [Operating cases](#) [► 855].

Explicit call of FB_init

● Explicit calling is not recommended

i The methods FB_init, FB_reinit and FB_exit are system functions that are called implicitly at different times (further information on this can be found under [Operating cases](#) [► 855]). Explicitly calling these methods can have inadvertent consequences and is therefore not recommended.

In principle, the method FB_init can also be called explicitly. However, this is not recommended.

In the case of an explicit call, not only is the explicit initialization code of the FB_init method executed again, but also the implicit initialization is repeated. For example, all variables are re-initialized, both in the case of automatic zero initialization and in the case of individual value initialization.

The effect of the latter is that both initialization steps are also performed for the FB instances declared in this instance. This procedure continues over the further instance levels contained.

If FB instances are re-initialized without being de-initialized first, this can lead to problems in the further program sequence, for example if the FB_init method contains dynamic memory allocations or an instance counter.

As the FBs are not used here for their intended purpose and it may not be possible to estimate the problematic consequences, we expressly advise you not to call FB_init methods explicitly.

Interface of the method FB_init

```
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
```

Through the evaluation of the FB_init method parameters you can distinguish between the operating cases and adapt the implementation if necessary. (See [Operating cases \[▶ 855\]](#))

Method parameters	(first/new) download	Online Change	TwinCAT restart (without new download)
bInitRetains	TRUE	FALSE	FALSE
bInCopyCode	FALSE	TRUE	FALSE

You can declare additional method inputs in an FB_init method. In this case, you have to set these inputs in the declaration of the function block instance.

Return value

Implicit calls

If the methods are called implicitly the return value will not be evaluated by the system. Even if you adjust the return value, it will not be evaluated with an implicit call.

Explicit calls

If the methods are called explicitly you can evaluate the return value. You can thus return a meaningful return value.

FB_init with derived function blocks

If a function block is derived from another function block, then the FB_init method of the basic function block is automatically executed for this function block. If the FB_init method of the derived function block is explicitly added, it is executed following the FB_init method of the basic function block (see [Behavior with derived function blocks \[▶ 859\]](#)).

If the FB_init method of the derived function block is to be present in explicit form, it must define the same parameters as the FB_init method of the basic function block. Further parameters can be added, in order to set up a special initialization for the derived instance.

i The FB_init method cannot be compared with the construct of a constructor, as it is known from C#, C++ or Java, since a function block in the PLC does not require a constructor to initialize its declared variables. This takes place implicitly or explicitly in the declaration lines.

The resulting consequences for function blocks that extend other function blocks are described in section [Behavior with derived function blocks \[▶ 859\]](#).

Samples for the assignment of additional FB_init parameters

Sample 1:

Method FB_SerialDevice.FB_init

```
METHOD PUBLIC 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 c
```

```
hange)
    nComNum      : INT; // additional input: number of the COM-interface that is to be observed
END_VAR
```

Declaration of the function block FB_SerialDevice:

```
fbCom1 : FB_SerialDevice(nComNum := 1);
fbCom0 : FB_SerialDevice(nComNum := 0);
```

Sample 2:

Function block FB_Sample:

```
FUNCTION_BLOCK FB_Sample
VAR
    nStartValue : INT;
END_VAR
```

Method FB_Sample.FB_init:

```
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 c
hange)
    nValue       : INT; // parameter to initialize the start value
END_VAR
```

```
nStartValue := nValue;
```

Declaration of the function block FB_Sample:

```
PROGRAM MAIN
VAR
    fbSample1 : FB_Sample(123); // => fbSample1.nStartValue is set to 123
    fbSample2 : FB_Sample(456); // => fbSample2.nStartValue is set to 456
END_VAR
```

Sample 3:

In the following sample, an input variable and a property are initialized by a function block that has an FB_init method [▶ 850](#) with an additional parameter.

Function block FB_Sample:

```
FUNCTION_BLOCK FB_Sample
VAR_INPUT
    nInput      : INT;
END_VAR
VAR
    nLocalInitParam : INT;
    nLocalProp      : INT;
END_VAR
```

Method FB_Sample.FB_init:

```
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)
    nInitParam   : INT;
END_VAR
```

```
nLocalInitParam := nInitParam;
```

Property FB_Sample.nMyProperty and the associated Set function:

```
PROPERTY nMyProperty : INT
nLocalProp := nMyProperty;
```

Program MAIN:

```
PROGRAM MAIN
VAR
    fbSample : FB_Sample(nInitParam := 1) := (nInput := 2, nMyProperty := 3);
    aSample  : ARRAY[1..2] OF FB_Sample[(nInitParam := 4), (nInitParam := 7)]
              := [(nInput := 5, nMyProperty := 6), (nInput := 8, nMyProperty := 9)];
END_VAR
```


Initialization result:

- fbSample
 - nInput = 2
 - nLocalInitParam = 1
 - nLocalProp = 3
- aSample[1]
 - nInput = 5
 - nLocalInitParam = 4
 - nLocalProp = 6
- aSample[2]
 - nInput = 8
 - nLocalInitParam = 7
 - nLocalProp = 9

See also:

- [Attribute 'call_after_init' \[► 796\]](#)
- [Operating cases \[► 855\]](#)

16.12.2 FB_reinit

If required, you have to implement FB_reinit explicitly. If this method is present, it is called automatically after the instance of the corresponding function block has been copied (implicit call). This happens during an Online Change after changes to the function block declaration (signature change) to reinitialize the new instance block.

● Explicit calling is not recommended

I The methods FB_init, FB_reinit and FB_exit are system functions that are called implicitly at different times (further information on this can be found under [Operating cases \[► 855\]](#)). Explicitly calling these methods can have inadvertent consequences and is therefore not recommended.

● Automatic core dump on exception in FB_init/FB_reinit/FB_exit

I If an exception error occurs within the FB_init/FB_reinit/FB_exit code, e.g. due to a programming error, the runtime system automatically stores a core dump on the target system (from TC3.1 Build 4024.25). This core dump is stored as a *.core file in the boot folder of the target system (by default under `C:\TwinCAT\3.1\Boot\Plc`) and can be used for the search of causes.

For more information on loading a core dump, see: [Error analysis with core dump \[► 247\]](#)

Interface of the method FB_reinit

```
METHOD FB_reinit : BOOL
```

Return value

Implicit calls

If the methods are called implicitly the return value will not be evaluated by the system. Even if you adjust the return value, it will not be evaluated with an implicit call.

Explicit calls

If the methods are called explicitly you can evaluate the return value. You can thus return a meaningful return value.

FB_reinit with derived function blocks

To reinitialize the basic implementation of the function block, FB_reinit must be called explicitly for the basic function block (via `SUPER^.FB_reinit()`). The return value can be evaluated.

16.12.3 FB_exit

If required, you have to implement FB_exit explicitly. If this method is present it is called automatically (implicitly) before the code of the function block instance is removed by the controller (e.g. even if TwinCAT is switched from Run mode to Configuration mode).

● Explicit calling is not recommended

i The methods FB_init, FB_reinit and FB_exit are system functions that are called implicitly at different times (further information on this can be found under [Operating cases \[▶ 855\]](#)). Explicitly calling these methods can have inadvertent consequences and is therefore not recommended.

● Automatic core dump on exception in FB_init/FB_reinit/FB_exit

i If an exception error occurs within the FB_init/FB_reinit/FB_exit code, e.g. due to a programming error, the runtime system automatically stores a core dump on the target system (from TC3.1 Build 4024.25). This core dump is stored as a *.core file in the boot folder of the target system (by default under `C:\TwinCAT\3.1\Boot\Pic`) and can be used for the search of causes.

For more information on loading a core dump, see: [Error analysis with core dump \[▶ 247\]](#)

Interface of the FB_exit method

```
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
```

Through the evaluation of the FB_exit method parameter you can distinguish between the operating cases and adapt the implementation if necessary. (see [Operating cases \[▶ 855\]](#))

Method parameters	(first/new) download	Online Change
bInCopyCode	FALSE	TRUE

Return value

Implicit calls

If the methods are called implicitly the return value will not be evaluated by the system. Even if you adjust the return value, it will not be evaluated with an implicit call.

Explicit calls

If the methods are called explicitly you can evaluate the return value. You can thus return a meaningful return value.

FB_exit with derived function blocks

If a function block is derived from another function block, then the FB_exit method of the basic function block is automatically executed for this function block. If the FB_exit method of the derived function block is explicitly added, this is executed first and then the FB_exit method of the basic function block (see [Behavior with derived function blocks \[▶ 859\]](#)).

See also:

- [Attribute 'no-exit' \[▶ 815\]](#)

16.12.4 Operating cases

The methods `FB_init`, `FB_reinit` and `FB_exit` are called implicitly at different times. You can find information on these different operating cases in the following:

- Operating case "First download"
- Operating case "New download"
- Operating case "Online Change"

The call behavior of the methods differs depending on the operating case. Through the querying of the method parameters "bInCopyCode" and "bInitRetains" by `FB_init` and `FB_exit` you can distinguish between the operating cases within the methods. This enables you to adapt the implementation to the respective operating case.

Operating case "First download"	Operating case "New download"	Operating case "Online Change"
<ol style="list-style-type: none"> 1. <code>FB_init</code> (implicit and explicit initialization code) 2. explicit external variable initialization via instance declaration of the function block 3. Method declared with attribute 'call_after_init' 	<ol style="list-style-type: none"> 1. <code>FB_exit</code> 2. <code>FB_init</code> (implicit and explicit initialization code) 3. explicit external variable initialization via instance declaration of the function block 4. Method declared with attribute 'call_after_init' 	<ol style="list-style-type: none"> 1. <code>FB_exit</code> 2. <code>FB_init</code> (implicit and explicit initialization code) 3. explicit external variable initialization via instance declaration of the function block 4. Method declared with attribute 'call_after_init' 5. Copy procedure 6. <code>FB_reinit</code>
<p>Method parameters:</p> <pre>FB_init(bInitRetains := TRUE, bInCopyCode := FALSE);</pre>	<p>Method parameters:</p> <pre>FB_exit(bInCopyCode := FALSE); FB_init(bInitRetains := TRUE, bInCopyCode := FALSE);</pre>	<p>Method parameters:</p> <pre>FB_exit(bInCopyCode := TRUE); FB_init(bInitRetains := FALSE, bInCopyCode := TRUE);</pre>

Operating case "First download"

When downloading a PLC project to a controller that is in the delivery state, the memory locations of all variables must be set to the desired initial state through initialization. In the process, the data areas of function block instances will be filled with the desired values. Through the explicit implementation of `FB_init` [► 850] for function blocks you can purposefully influence the initialization in this situation.

By evaluating the `FB_init` method parameters "bInitRetains" (TRUE) and "bInCopyCode" (FALSE) you can detect this operating case unambiguously. (See also operating case "Online Change")

i The `FB_init` method cannot be compared with the construct of a constructor, as it is known from C#, C++ or Java, since a function block in the PLC does not require a constructor to initialize its declared variables. This takes place implicitly or explicitly in the declaration lines.

The resulting consequences for function blocks that extend other function blocks are described in section Behavior with derived function blocks [► 859].

Attribute 'call_after_init' as an alternative to `FB_init`

Following the call of `FB_init` and prior to the first cycle of the tasks of a PLC project, the external initial assignments are processed within the declaration of the function block instance.

```
fbTimer : TON := (PT := T#500MS);
```

Such assignments are only executed after `FB_init` has been called. If TON had an `FB_init` method, the assigned time value of T#500MS would not be known in this method.

To control the effects of these assignments, you can assign the attribute {attribute 'call_after_init'} [► 796] to a method of a function block. You must insert the attribute both above the declaration part of the corresponding method and above the declaration part of the function block body.

The method is called after the processing of the external initial assignments and prior to the start of the tasks of a PLC project and can thus react appropriately to the user's specifications (with the external initialization).

Sample

```
{attribute 'call_after_init'}
FUNCTION_BLOCK FB_Sample
VAR_INPUT
    nInput1    : INT;
    nInput2    : INT := 100;
END_VAR

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

{attribute 'call_after_init'}
METHOD MyCallAfterInit

PROGRAM MAIN
VAR
    fbSample : FB_Sample := (nInput2 := 700);
END_VAR
```

During the download, the following calls are made one after the other:

1. FB_init	<p>Initialize instance</p> <p>1. implicit initialization code (implicit zero initialization and explicit internal value initialization of the variables)</p> <pre>nInput1 := 0; nInput2 := 100;</pre> <p>2. explicit initialization code (initialization code explicitly defined in FB_init)</p> <pre>fbSample.FB_init(bInitRetains := TRUE, bInCopyCode := FALSE);</pre> <p>By evaluating the FB_init method parameters, you can detect this operating case unambiguously. Via the FB_init method you can, for example, initialize variables with the help of additional initialization code or inform other parts of the PLC project of the position of certain variables in the memory.</p>
2. explicit external variable initialization via instance declaration of the function block	<p>Processing external initial assignments</p> <pre>nInput2 := 700;</pre> <p>If the input variables of the function block have been assigned values, they will be copied. The original value is retained in the case of variables to which no value has been assigned from outside.</p>
3. Method declared with attribute call_after_init	<p>Alternative initialization</p> <pre>fbSample.MyCallAfterInit();</pre> <p>In order to unambiguously detect this operating case you can copy the value of bInCopyCode beforehand into an auxiliary variable in FB_init and evaluate this auxiliary variable in the 'call_after_init' method. You can use the attribute as an alternative to FB_init or, for example, check the effects of the explicit external variable initialization. Make the implementation as independent as possible. The method can also be called from the PLC project at any time to restore a function block instance to its original state.</p>

Operating case "New download"

When downloading a PLC project again, a project already existing on the controller may be replaced. Therefore the memory space for the existing function blocks must first be released in a controlled manner. You can use the method [FB_exit](#) [▶ 854] for this. If this method has been created it will be called before the

old project is removed. Subsequently, the new project will be loaded to the controller and FB_init called. In FB_exit you can, for example, set external resources (with socket or file handles) to a defined state or release dynamically allocated memory (__NEW- or in this case __DELETE-Operator).

By evaluating the FB_exit method parameter "blnCopyCode" (FALSE), you can detect this operating case unambiguously.

Operating case "Online Change"

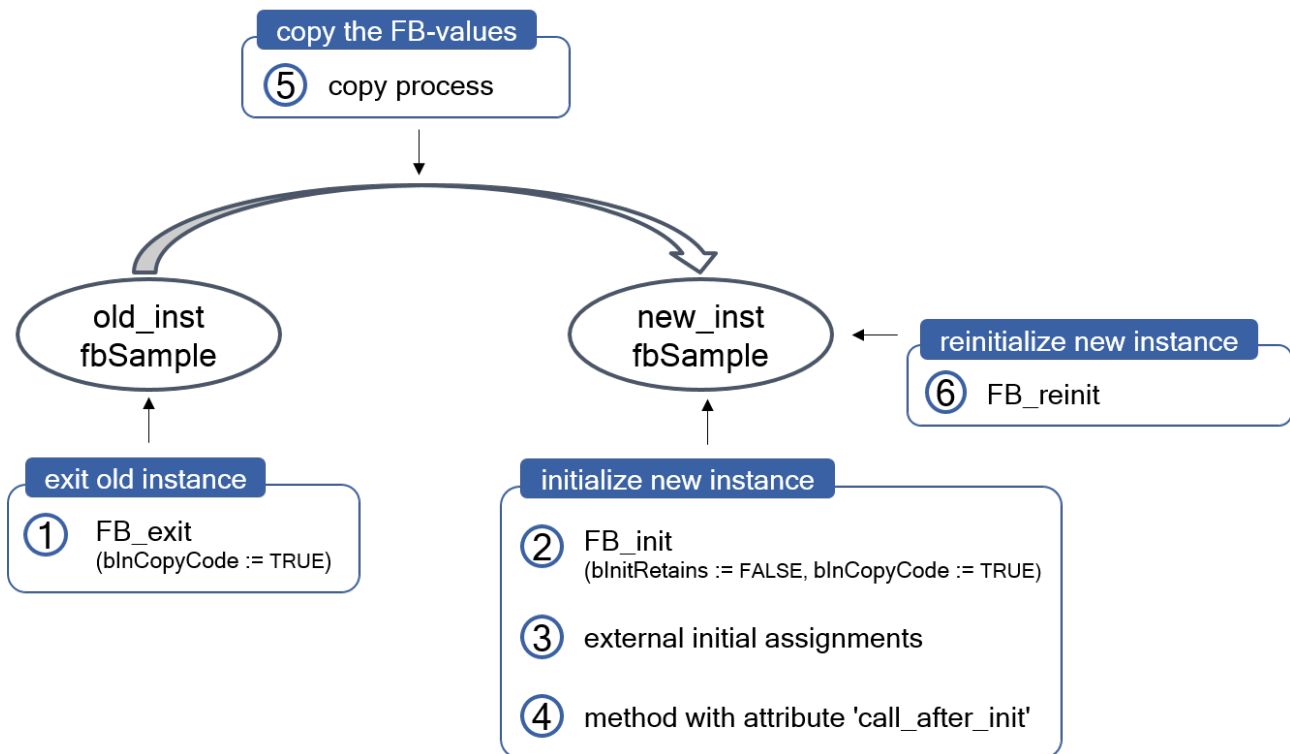
During an Online Change, you can influence the initialization of function block instances via the methods FB_exit, FB_init and FB_reinit. As part of the Online Change, the changes made to the PLC project during offline operation are tracked in the current control system. Therefore, the "old" instances of the function blocks are replaced by their "new" siblings as smoothly as possible.

If no changes were made in the declaration part of a function block before logging into the PLC project (i.e. changes were only made in the implementation), the data areas are not replaced. Only code blocks are replaced. The methods FB_exit, FB_init and FB_reinit are not called in this case.

i If you have made changes to the declaration of a function block that will lead to the copy process described above, you receive a message about "possible unintentional effects" during an Online Change. The message box **Details** contain a list of all instances to be copied.

In the code of the methods FB_init and FB_exit it is possible by evaluating the parameters "blnitRetains" (FALSE) and "blnCopyCode" (TRUE) to determine whether an Online Change is currently being executed that concerns the function block instances and is shifting them to a different memory location.

During the Online Change, the following calls take place one after the other:



1. FB_exit	<p>Exit old instance</p> <pre>old_inst.FB_exit(bInCopyCode := TRUE);</pre> <p>By evaluating the FB_exit method parameter, you can detect this operating case unambiguously.</p> <p>You can use the call of FB_exit when leaving the "old" instance to trigger certain cleanup tasks before copying. In this way, you can prepare the data for the next copy operation and influence the state of the "new" instance. Other parts of the PLC project can inform you about the imminent change of position in the memory.</p> <p>Pay particular attention to variables of type POINTER or REFERENCE. After the Online Change, these may no longer point to the desired storage locations. (Note: you can use the Attribute 'call on type change' [► 797] to react to the data type change of a referenced function block).</p> <p>Interface variables (INTERFACE) are treated separately by the compiler and adapted accordingly during the Online Change. External resources such as sockets, files or other handles can possibly be accepted unchanged by the new instance. In many cases they do not have to be treated separately during the Online Change. (See operating case "New download")</p>
2. FB_init	<p>Initialize new instance</p> <ol style="list-style-type: none"> implicit initialization code (implicit zero initialization and explicit internal value initialization of the variables) explicit initialization code (initialization code explicitly defined in FB_init) <pre>new_inst.FB_init(bInitRetains := FALSE, bInCopyCode := TRUE);</pre> <p>By evaluating the FB_init method parameters, you can detect this operating case unambiguously.</p> <p>The call of FB_init takes place after the copy operation and can be used to perform specific operations for the Online Change. For example, you can inform other parts of the PLC project about the new position of certain variables in the memory.</p>
3. explicit external variable initialization via instance declaration of the function block	<p>Processing external initial assignments</p> <pre>new_inst : <FB-Name> := (<Variable>:=<value>);</pre> <p>If the input variables of the function block have been assigned values, they will be copied. The original value is retained in the case of variables to which no value has been assigned from outside.</p>
4. Method declared with attribute call_after_init	<p>Alternative initialization</p> <pre>new_inst.<Methodenname der gekennzeichneten Methode>();</pre> <p>In order to unambiguously detect this operating case you can copy the value of bInCopyCode beforehand into an auxiliary variable in FB_init and evaluate this auxiliary variable in the 'call_after_init' method.</p> <p>For example, you can use the attribute as an alternative to FB_init.</p> <p>Make the implementation as independent as possible. The method can also be called from the PLC project at any time to restore a function block instance to its original state.</p>
5. Copy process	<p>Copy function block values (copy code)</p> <pre>copy(&old_inst, &new_inst);</pre> <p>Existing values are retained. To this end, they are copied from the old instance to the new instance.</p>
6. FB_reinit	<p>Reinitialize new instance</p> <pre>new_inst.FB_reinit();</pre> <p>This method is called after the copy operation. It sets the variables of the instance to defined values.</p> <p>For example, you can have variables at the "new" position in the memory initialized accordingly or inform other parts of the PLC project about the new position of certain variables in the memory.</p> <p>The implementation should always be handled independent of the Online Change. The method can also be called from the PLC project at any time to restore a function block instance to its original state.</p>



The `attribute {attribute 'no_copy'} [▶ 812]` can be used to prevent a specific variable of the function block from being copied during the Online Change. In this case the variable always retains the initial value.

See also:

- [Attribute 'call_after_init' \[▶ 796\]](#)
- [Attribute 'call_on_type_change' \[▶ 797\]](#)
- [Attribute 'no_copy' \[▶ 812\]](#)
- [Attribute 'no-exit' \[▶ 815\]](#)

16.12.5 Behavior with derived function blocks

When using the methods `FB_init`, `FB_reinit` and `FB_exit` in the context of derived function blocks, please note the following.

FB_init with derived function blocks

If a function block is derived from another function block, then the `FB_init` method of the basic function block is automatically executed for this function block. If the `FB_init` method of the derived function block is explicitly added, it is executed following the `FB_init` method of the basic function block (see [Behavior with derived function blocks \[▶ 859\]](#)).

If the `FB_init` method of the derived function block is to be present in explicit form, it must define the same parameters as the `FB_init` method of the basic function block. Further parameters can be added, in order to set up a special initialization for the derived instance.

FB_reinit with derived function blocks

To reinitialize the basic implementation of the function block, `FB_reinit` must be called explicitly for the basic function block (via `SUPER^.FB_reinit()`). The return value can be evaluated.

FB_exit with derived function blocks

If a function block is derived from another function block, then the `FB_exit` method of the basic function block is automatically executed for this function block. If the `FB_exit` method of the derived function block is explicitly added, this is executed first and then the `FB_exit` method of the basic function block (see [Behavior with derived function blocks \[▶ 859\]](#)).

Sample of the calling order in the case of derived function blocks

The function blocks `FB_Base`, `FB_Sub` and `FB_SubSub` are derived from each other. The following applies:

- `FB_Sub` EXTENDS `FB_Base`
- `FB_SubSub` EXTENDS `FB_Sub`

Situation:

- All three function blocks have their own `FB_init`, `FB_reinit` and `FB_exit` methods.
- The function block `FB_SubSub` is instantiated. An additional variable is added to the function block via Online Change.

Assumption - case 1:

- No reinitialization of the basic implementation is required. This means that the call `SUPER^.FB_reinit()` does not exist in the method `FB_SubSub.FB_reinit`.

Call sequence:

The call sequence for the methods `FB_exit`, `FB_init` and `FB_reinit` for the function block instance `fbSubSub` is then as follows:

1. Impliziter Aufruf von `FB_SubSub.FB_exit(TRUE)` für `fbSubSub`
2. Impliziter Aufruf von `FB_Sub.FB_exit(TRUE)` für `fbSubSub`
3. Impliziter Aufruf von `FB_Base.FB_exit(TRUE)` für `fbSubSub`
4. Impliziter Aufruf von `FB_Base.FB_init(FALSE, TRUE)` für `fbSubSub`
5. Impliziter Aufruf von `FB_Sub.FB_init(FALSE, TRUE)` für `fbSubSub`
6. Impliziter Aufruf von `FB_SubSub.FB_init(FALSE, TRUE)` für `fbSubSub`
7. Impliziter Aufruf von `FB_SubSub.FB_reinit()` für `fbSubSub`

Assumption – case 2:

- A reinitialization of the basic implementation is required. This means that the methods `FB_SubSub.FB_reinit` and `FB_Sub.FB_reinit` contain the call `SUPER^.FB_reinit()`.

Call sequence:

The call sequence for the methods `FB_exit`, `FB_init` and `FB_reinit` for the function block instance `fbSubSub` is then as follows:

1. Impliziter Aufruf von `FB_SubSub.FB_exit(TRUE)` für `fbSubSub`
2. Impliziter Aufruf von `FB_Sub.FB_exit(TRUE)` für `fbSubSub`
3. Impliziter Aufruf von `FB_Base.FB_exit(TRUE)` für `fbSubSub`
4. Impliziter Aufruf von `FB_Base.FB_init(FALSE, TRUE)` für `fbSubSub`
5. Impliziter Aufruf von `FB_Sub.FB_init(FALSE, TRUE)` für `fbSubSub`
6. Impliziter Aufruf von `FB_SubSub.FB_init(FALSE, TRUE)` für `fbSubSub`
7. Impliziter Aufruf von `FB_SubSub.FB_reinit()` für `fbSubSub`
8. Expliziter Aufruf von `FB_Sub.FB_reinit()` für `fbSubSub`
9. Expliziter Aufruf von `FB_Base.FB_reinit()` für `fbSubSub`

17 Reference User Interface

By default, the main commands are available in the TwinCAT user interface. To customize the menu configuration, select **Customize** from the **Tools** menu.

See also:

- [Customizing the user interface](#)

17.1 File

17.1.1 Archiving options

The TwinCAT development environment offers three different archiving file types for storing a TwinCAT project, e.g. for archiving purposes or for passing it on to colleagues: tnzipe, tszip and tpzip.

The type of archive file you should use depends on which projects you want to store in the archive folder.

Archiving file type	Focus	Contents	Commands
tnzip	Solution [▶ 861]	The *.tnzip archive folder contains all TwinCAT project types included in the solution. These can be TwinCAT, TwinCAT HMI, Scope or Connectivity projects.	Build [▶ 861] Open [▶ 862]
tszip	TwinCAT project [▶ 862]	The *.tszip archive folder contains the TwinCAT project to be archived.	Build [▶ 1064] Open [▶ 862]
tpzip	PLC project [▶ 863]	The *.tpzip archive folder contains the PLC project to be archived.	Build [▶ 863] Open [▶ 863]

17.1.1.1 Solution

- Creating a *.tnzip TwinCAT solution archive: [Command Save <solution name> as Archive... \[▶ 861\]](#)
- Open *.tnzip TwinCAT solution archive: [Command Open Solution from Archive \[▶ 862\]](#)

17.1.1.1.1 Command Save <solution name> as Archive...

Function: The command opens the standard dialog for saving a file as an archive. The solution can be stored as a *.tnzip archive under the desired path.

Call: File menu, context menu

Requirement: The solution is selected in the **Solution Explorer**.

Content of *.tnzip	The *.tnzip archive folder contains all TwinCAT project types included in the solution. These can be TwinCAT, TwinCAT HMI, Scope or Connectivity projects.
Command for opening	A tnzipe archive can be reopened with the following command: Command Open Solution from Archive [▶ 862] .
Note on PLC projects	If the solution contains one or more PLC projects, the files and folders stored in the archive folder for those PLC projects will depend on the PLC project settings of the respective project. Settings tab [▶ 926]

17.1.1.1.2 Command Open Solution from Archive

Function: The command extracts a TwinCAT solution archive *.tzip.

Call: Menu **File > Open**


Executing the command opens the **Open** dialog. Select the archive file from the file system and confirm the dialog. The **Select Folder for new Solution** dialog opens. Select a folder for storing the extracted solution files.

Content of *.tzip	The *.tzip archive folder contains all TwinCAT project types included in the solution. These can be TwinCAT, TwinCAT HMI, Scope or Connectivity projects.
Command for creating	A tzip archive can be created with the following command: Command Save <solution name> as Archive... [▶ 861]
Note on PLC projects	If the solution contains one or more PLC projects, the files and folders stored in the archive folder for those PLC projects will depend on the PLC project settings of the respective project. Settings tab [▶ 926]

17.1.1.2 TwinCAT project

- Creating a *.tzip TwinCAT project archive: Command Save <TwinCAT project name> as Archive... [▶ 1064]
- Open a *.tzip TwinCAT project archive: Command Project/Solution (Open Project/Solution) [▶ 862]

17.1.1.2.1 Command Project/Solution (Open Project/Solution)

Symbol: 

Hotkey: **[Ctrl] + [Shift] + [O]**

Function: The command opens the default dialog for opening a file. Here you can browse the file system for a TwinCAT project file and open it in the development system.

Call: Menu **File > Open**

Open Project dialog

File type	Selection list for filtering the file type <ul style="list-style-type: none"> • Files of all supported formats can be opened.
Options	<ul style="list-style-type: none"> • Add (Use this option only to add a measurement project to a solution, for example. Do not use the option to add multiple TwinCAT projects to a solution.) • Close solution
Open	TwinCAT opens the selected project file. If necessary, it is converted first.

TwinCAT *.tzip project archive

Content of *.tzip	The *.tzip archive folder contains the TwinCAT project to be archived.
Command for creating	A tzip archive can be created with the following command: <u>Command Save <TwinCAT project name> as Archive...</u> [▶ 1064]
Note on PLC projects	If the TwinCAT project contains one or more PLC projects, the files and folders stored in the archive folder for those PLC projects will depend on the PLC project settings of the respective project. <u>Settings tab</u> [▶ 926]

See also:

- PLC documentation: [Your first TwinCAT 3 PLC project](#) [[▶ 24](#)]
- PLC documentation: [Creating a standard project](#) [[▶ 54](#)]

17.1.1.3 PLC project

- Creating a *.tzip PLC project archive: [Command Save <PLC project name> as archive...](#) [[▶ 863](#)]
- Open *.tzip PLC project archive: [Command Add Existing Item \(Project\)](#) [[▶ 863](#)]

17.1.1.3.1 Command Save <PLC project name> as archive...

Function: The command opens the standard dialog for saving a file as an archive. The PLC project can be stored as a *.tzip archive under the desired path.


Call: File menu, context menu

Requirement: The TwinCAT PLC project (<PLC project name>) is selected in the **Solution Explorer**.

The files and folders in the archive folder depend on the PLC project settings.

Content of *.tzip	The *.tzip archive folder contains the PLC project to be archived.
Command for opening	A tzip archive can be reopened with the following command: <u>Command Add Existing Item (Project)</u> [▶ 863]
Note on PLC projects	The files and folders stored in the archive folder for the PLC project depend on the PLC project settings of this PLC project. <u>Settings tab</u> [▶ 926]

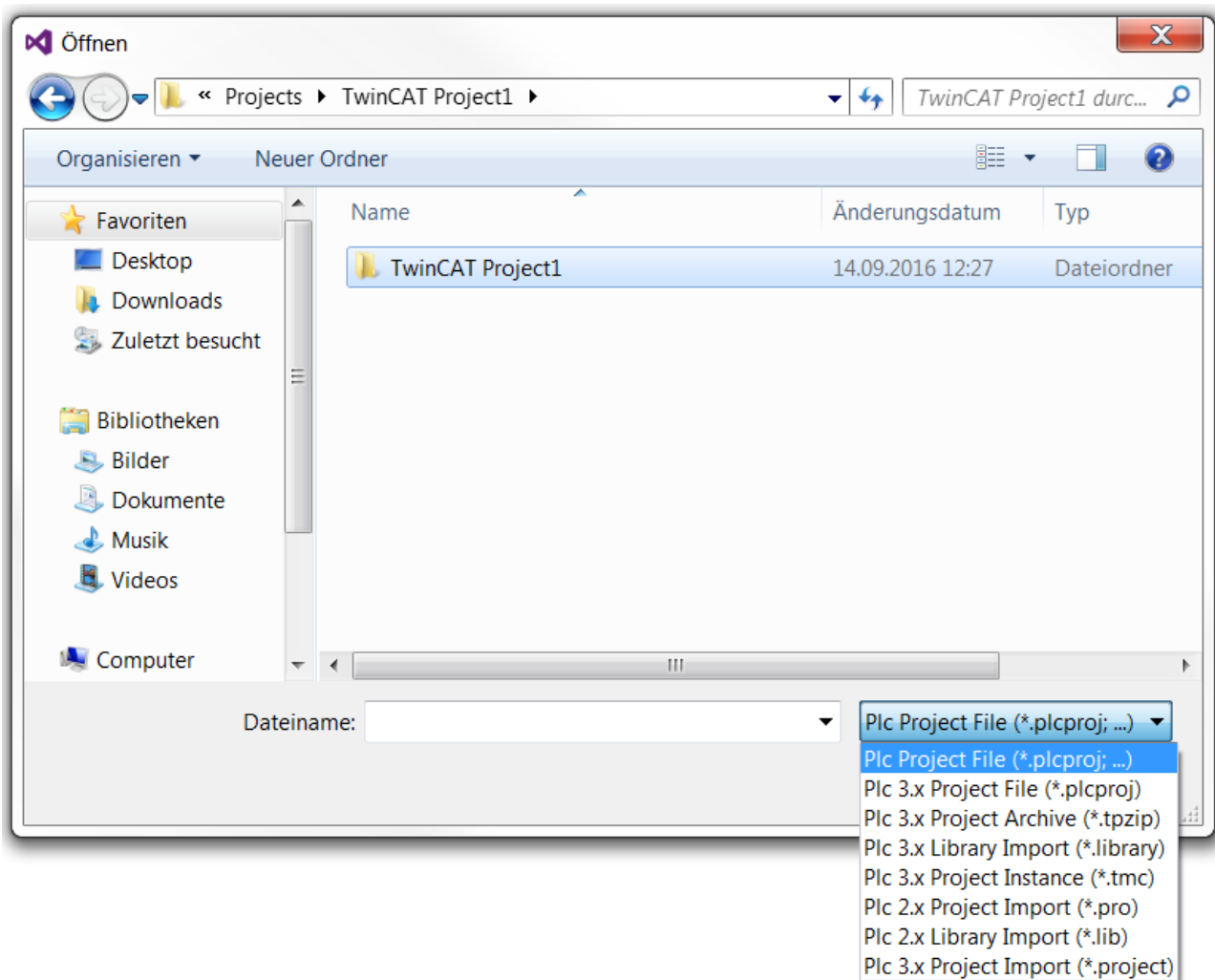
17.1.1.3.2 Command Add Existing Item (Project)

Symbol: 

Function: The command opens the standard browser dialog, which can be used to search for a PLC project file and open it in the programming system. If a suitable converter is installed, you can open projects in a different format.

Call: Project menu or PLC object context menu in the **Solution Explorer**

Requirement: The PLC node is selected in the TwinCAT project tree.



File type	<p>By default, you can set the filter to one of the following file types:</p> <ul style="list-style-type: none"> • PLC 3.x Project file (*.PLCproject): TwinCAT 3 PLC projects with the extension ".PLCproject" • PLC 3.x Project archive (*.tpzip): TwinCAT 3 PLC project archives with the extension ".tpzip" <ul style="list-style-type: none"> ◦ See also: Command Save <PLC project name> as archive... [► 863] • PLC 3.x Library import (*.library): TwinCAT 3 PLC libraries with the extension ".library" • PLC 2.x Project file (*.pro): TwinCAT 2 PLC projects with the extension ".pro" • PLC 2.x Import library (*.lib): TwinCAT 2 PLC libraries with the extension ".lib" • PLC 3.x Project import (*.PLCproject): PLC projects with the extension ".project"
Open	The selected project file is opened or converted and then opened.

***.tpzip PLC project archive**

Content of *.tpzip	The *.tpzip archive folder contains the PLC project to be archived.
Command for creating	<p>A tpzip archive can be created with the following command:</p> <p>Command Save <PLC project name> as archive... [► 863]</p>
Note on PLC projects	<p>The files and folders stored in the archive folder for the PLC project depend on the PLC project settings of this PLC project.</p> <p>Settings tab [► 926]</p>

Possible scenarios when opening a PLC project

The following scenarios are possible when you open a project:

1. [Another project is still open. \[▶ 865\]](#)
2. [The project was saved with an older version of TwinCAT 3. \[▶ 865\]](#)
3. [The project was not saved with TwinCAT 3. \[▶ 865\]](#)
4. [The project was not terminated properly and "Save automatically" was enabled. \[▶ 866\]](#)
5. [The project is read-only. \[▶ 866\]](#)
6. [It is a library that is installed in a library repository and retrieved from it. \[▶ 867\]](#)

1. Another project is still open.

You are asked if the other project should be saved and closed.

2. The project was saved with an older version of TwinCAT 3.

If the file format differs because the open project was saved with an older version of TwinCAT 3, there are two options:

- If the project cannot be saved in the format of the currently used programming system, you must update it to continue working on the project. The expression that appears at this point: **The changes you made...** refers to internal tasks of various components while the project is loaded.
- If the project can still be saved in the previous format, you can decide whether to update or retain the format. If you decide to retain the format, data loss may occur. If you decide to update the format, the project can no longer be opened with the old version of the programming system.

In addition to the file format, the versions of the explicitly inserted libraries, the visualization profile and the compiler version of the opening project may differ from those installed with the current programming system.

If newer versions are installed on the current programming system, the **Project Environment** dialog opens automatically, where you can update the versions. If no update is made at this point, this can be done later at any time via the **Options > Project Environment** dialog.

● Note the compiler version

i If a project is opened that was created with an older version of the programming system and for which the latest compiler version is set in the project settings, while the project environment setting for the compiler version is set to **Do not update** in the new programming system, the compiler version that was used last in the old project continues to be used (i.e. not the "Current" version in the new environment).

3. The project was not saved with TwinCAT 3.

Case 1)

If you set the file filter when selecting the project to be opened and an appropriate converter is available, the converter is used automatically and the project is brought into the current format. The conversion is converter-specific. Usually, you are prompted to define the handling of referenced libraries or device references.

● TwinCAT 3 converter

i Adaptation of a TwinCAT PLC control project to the TwinCAT 3 syntax can only be successful during import if the converter is able to compile the project without errors.

If you have set the **All Files** option when selecting the project to be opened, no converter is enabled and the **Convert Project** dialog opens. In the dialog, you need to explicitly trigger the conversion of the project by selecting one of the options.

- **Convert to the current format:** Select the converter you want to use from the selection list (application for conversion). After the conversion, the project can no longer be opened in the old version.
- **Create a new project and add a specific device:** (Not yet implemented)

i TwinCAT 2.x PLC Control project options

The project directory path set in the TwinCAT 2.x PLC control project options and the project information are adopted in the **Project Information** dialog.

Case 2)

If libraries are integrated in the project for which "conversion mapping" has not yet been stored in the library options, the **Converting a library reference** dialog appears, in which you can define how this reference should be converted:

- **Convert and install the library:** If you select this option, the referenced library is converted to the new format and remains referenced in the project. It is automatically installed in the library repository under the **Other** category and continues to be used. If the library does not have the project information (title, version) required for an installation, you will be prompted to enter it in the **Enter Project Information** dialog.
- **Use the following library, which is already installed:** If you select the options, the referenced library is replaced by another one that is already installed on the local system. Use the **Select** button to open the **Select...** dialog. Here you can select the desired version of one of the installed libraries. This corresponds to the configuration of the version handling in the **Library Properties** dialog. An asterisk ("*") means that, as a rule, the latest version of the library available on the system is used in the project. The list of available libraries is structured in the same way as in the **Library Repository** dialog. You can sort the list by company and category.
- **Ignore the library. The reference will not appear in the converted project:** If you enable this option, the library reference is removed. The library is then no longer included in the converted project.
- **Use this mapping in future if this library is present:** If you enable this option, the settings made here in the dialog will also be applied to future project conversions, as soon as the respective library is referenced.

In the converted project, the library references are defined in the Global Library Manager in the Solution Explorer. After the conversion of the library references, the project conversion continues with the **Open Project** dialog, as described above.

For general information on library management, see section "Using libraries" in the PLC documentation.

Case 3)

When you open a TwinCAT 2.x PLC Control project that references a device (target system) for which no "conversion mapping" has yet been defined in the TwinCAT 2.x PLC Control converter options, the **Device Conversion** dialog opens, in which you can specify whether and how the old device references are to be replaced by more recent ones. The device originally used is displayed. Choose one of the following options:

- **Use the following already installed device:** Click the **Select** button to open the **Select target system** dialog, in which you can select one of the devices currently installed on the system. This device is then inserted in the **Solution Explorer** of the converted project, instead of the old one. Select the option **Select a target system...** to select one of the devices listed. The list of available devices is structured in the same way as in the **Device Repository** dialog. You can sort the list by manufacturer or by category.
- **Ignore the device. No application-specific objects will be available:** If you enable this option, no entry for the device is created in the **Solution Explorer** of the new project, i.e. the device is ignored during the conversion, and no application-specific objects, such as the task configuration, are applied.
- **Save this assignment for future reference:** If you select this option, all settings of the dialog, i.e. the displayed "conversion mapping" for the device, are saved in the TwinCAT 2.x PLC Control Converter options and applied to future conversions.

4. The project was not terminated properly and "Save automatically" was enabled.

If the **Auto Save** function was enabled in the **Load and Save** options and TwinCAT 3 PLC was not terminated regularly after the last modification of the project without saving, the **Auto Save Backup** dialog opens for handling the backup copy.

5. The project is read-only.

If the project to be opened is read-only, you are asked whether you want to open the project in write-protected mode or whether you want to unlock it.


6. It is a library that is installed in a library repository and retrieved from it.

An error message is displayed if you try and open a library project that is installed in a library repository. You cannot open a library project using this path. After closing the dialog with **OK**, the project name appears in the title bar of the user interface. An asterisk ("*") after the name indicates that the project has been modified since it was last saved.

See also:

- PLC documentation: [Open a TwinCAT 3 PLC project \[▶ 57\]](#)
- PLC documentation: [Open a TwinCAT 2 PLC project \[▶ 57\]](#)

17.1.2 Command Project... (Create new TwinCAT project)

Symbol: 

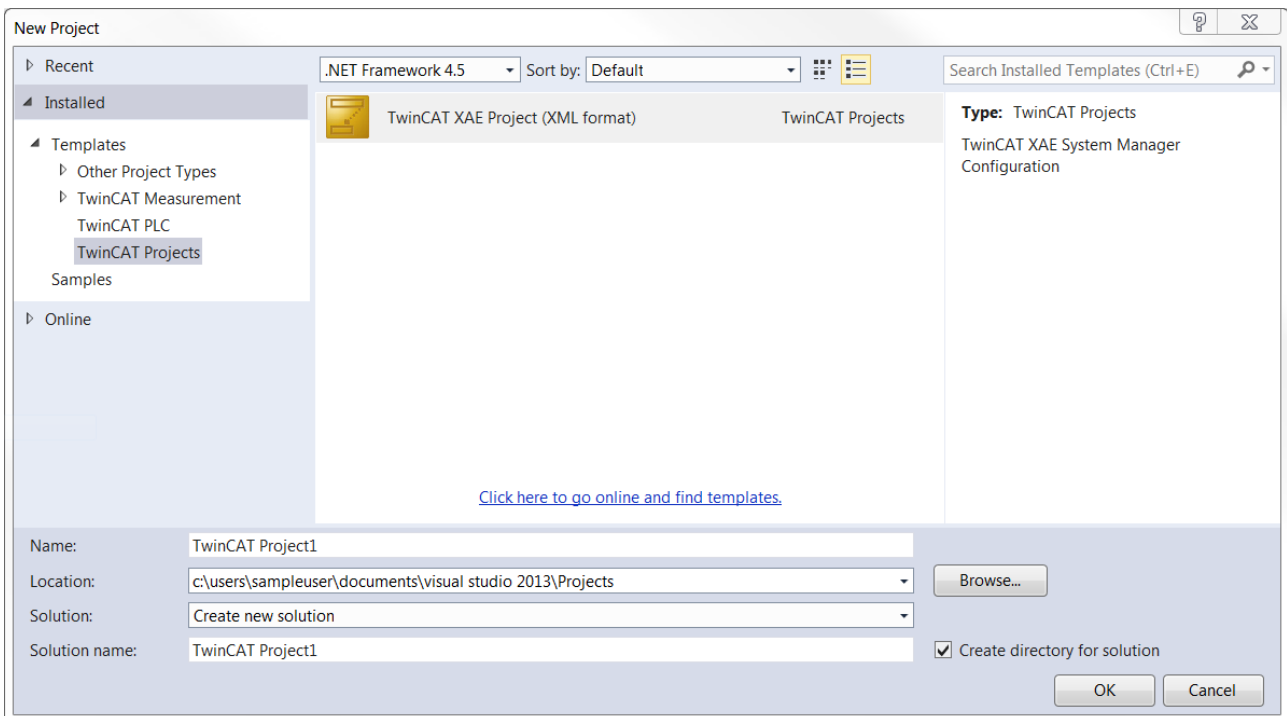
Hotkey: **[Ctrl] + [Shift] + [N]**

Function: The command opens the **New Project** dialog for creating a new TwinCAT project file.

Call: Menu **File > New**

New Project dialog

Depending on the selected template, you will obtain a project that automatically has a certain number of objects.



Categories

Recent	Shows the last used project template.
Installed > Templates	Shows the TwinCAT project templates: <ul style="list-style-type: none"> • Other Project Types • TwinCAT Measurement • TwinCAT PLC • TwinCAT Projects
Online	Not relevant

Templates

Category Other Project Types	
Visual Studio Solutions	Empty Visual Studio Solution
Category TwinCAT Measurement	
BodePlot	Bode Plot
Scope	Scope YT Project Scope YT NC Project Scope YT Project with Reporting Scope XY Project Scope XY Project with Reporting
Category TwinCAT PLC	
	TwinCAT PLC Project
Category TwinCAT Projects	
	TwinCAT XAE Project

Name	Name of the project to be created. A default name appears, depending on the template. The numeric supplement ensures the uniqueness of the name in the file system. You can change the file name according to the file path conventions of the operating system. Dots are not allowed in the name. TwinCAT automatically adds the file extension that matches the selected template.
Location	Location for the new project file. The Browse... button opens a dialog for browsing the file system. The combobox field shows the history of previously entered paths.
Solution	<ul style="list-style-type: none"> • Create new solution • Add • Create in new instance
Create directory for solution	<input checked="" type="checkbox"/> Solution directory is created.
Solution name	Name of the solution. By default, the TwinCAT project name is automatically adopted.
OK	TwinCAT opens a new project.

See also:

- PLC documentation: [Your first TwinCAT 3 PLC project \[▶ 24\]](#)
- PLC documentation: [Creating a standard project \[▶ 54\]](#)

17.1.3 Command Project/Solution (Open Project/Solution)

Symbol: 

Hotkey: **[Ctrl] + [Shift] + [O]**

Function: The command opens the default dialog for opening a file. Here you can browse the file system for a TwinCAT project file and open it in the development system.

Call: Menu **File > Open**

Open Project dialog

File type	Selection list for filtering the file type <ul style="list-style-type: none"> Files of all supported formats can be opened.
Options	<ul style="list-style-type: none"> Add (Use this option only to add a measurement project to a solution, for example. Do not use the option to add multiple TwinCAT projects to a solution.) Close solution
Open	TwinCAT opens the selected project file. If necessary, it is converted first.

TwinCAT *.tzip project archive

Content of *.tzip	The *.tzip archive folder contains the TwinCAT project to be archived.
Command for creating	A tzip archive can be created with the following command: Command Save <TwinCAT project name> as Archive... [▶ 1064]
Note on PLC projects	If the TwinCAT project contains one or more PLC projects, the files and folders stored in the archive folder for those PLC projects will depend on the PLC project settings of the respective project. Settings tab [▶ 926]

See also:

- PLC documentation: [Your first TwinCAT 3 PLC project \[▶ 24\]](#)
- PLC documentation: [Creating a standard project \[▶ 54\]](#)

17.1.4 Command Open Project from Target

Function: The command loads a project from the target system.

Call: Menu **File > Open**

Requirement: The network path to the target system must be configured.

Executing the command opens an overview of all devices on the network. Select the target system from this overview. The **Select Folder for new Solution** dialog opens.

17.1.5 Command New Project... (Add new TwinCAT project)

Function: The command opens the **New Project** dialog for creating a further TwinCAT project file in the solution.

Call: Menu **File > Add**

Requirement: A TwinCAT project is opened.



Use this command only to add a measurement project to a solution, for example. Do not use this command to add multiple TwinCAT projects to a solution. This function is currently not supported by TwinCAT.

17.1.6 Command Existing Item... (Add existing TwinCAT project)

Function: The command opens the **Add Existing Item** dialog for adding a TwinCAT project file to the solution.

Call: Menu **File > Add**

Requirement: A TwinCAT project is opened.



Use this command only to add a measurement project to a solution, for example. Do not use this command to add multiple TwinCAT projects to a solution. This function is currently not supported by TwinCAT.

17.1.7 Command Recent Projects and Solutions

Function: The command opens the list of recently used projects, from which you can select a project to open.

Call: Menu **File**

See also:

- PLC documentation: [Creating and configuring a project \[► 54\]](#)

17.1.8 Command Save All

Symbol:

Function: The command saves all objects of the TwinCAT project.

Call: **File** menu, standard toolbar options

17.1.9 Command Save

Symbol:

Hotkey: **[Ctrl] + [S]**

Function: The command saves the solution, the TwinCAT project, the TwinCAT PLC project or a selected PLC object (Main, GVL, ...) under the current name.

Call: **File** menu, standard toolbar options

Requirement: The solution, the TwinCAT project object, the PLC project object (<PLC project name> Project) or the PLC object to be saved is selected in the **Solution Explorer**.

Save object

The command saves the object under the current name. If the object has been changed since the last save, the "disk" icon for the object is red, and the name in the title bar of the editor for the open object is marked with an asterisk ("*").

See also:

- [Command Save <TwinCAT project name> as \[► 870\]](#)

17.1.10 Command Save <Solution name> as

Function: The command opens the default dialog for saving a file. The solution can be stored under the desired path. By default, the file type UTF-8 solution file (*.sln) is selected.

Call: Menu **File**

Requirement: The solution is selected in the **Solution Explorer**.

17.1.11 Command Save <TwinCAT project name> as

Function: The command opens the default dialog for saving a file. The project can be stored under the desired path and file type. By default, the file type TwinCAT XAE project (*.tsproj) is selected.

Call: Menu **File**

Requirement: The TwinCAT project object is selected in the **Solution Explorer**.

Note that during this saving operation only the *.tsproj file, for example, is created in a different location. The referenced projects and the objects contained in them are not stored at this new location (for example, a PLC project that is integrated in the TwinCAT 3 project and its objects).

See also:

- [Command Save \[► 870\]](#)
- PLC documentation: Library creation

17.1.12 Command Save <PLC project name> as

Function: The command opens a dialog in which a destination directory for the PLC project file can be specified. The PLC project objects and the .plcproj file are stored in the selected directory.

Call: PLC project object context menu


Requirement: The PLC project object (<PLC project name>) is selected in the **Solution Explorer**.

17.1.13 Command Create disassembly file

Function: The command creates a <project name>.asm disassembly file from the current project and saves it in the file directory in the project folder.

Call: PLC menu > Create disassembly file

17.1.14 Command Send by E-Mail...

Symbol: 

Function: The command starts the email program that is set in the system and opens a new email with the archive file of the selected project as an attachment.

Call: **File** menu, context menu

17.1.15 Command Close Solution

Symbol: 

Function: The command closes the currently open project. TwinCAT remains open.

Call: **File** menu or implicitly when opening a new/different project while a project is still open.

If the project contains unsaved changes, a query appears asking if the project should be saved.

If you have not yet explicitly saved the project, a query is displayed to confirm whether you want to delete the project files.

See also:

- PLC documentation: [Creating and configuring a PLC project \[► 54\]](#)

17.1.16 Command Close

Function: The command closes the open editor.

Call: Menu **File**

Requirement: The editor to be closed is active, or the object is selected in the PLC project tree.

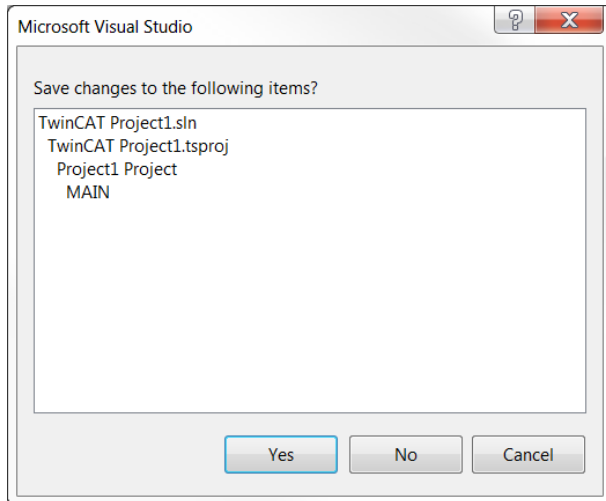
17.1.17 Command Exit

Symbol: 


Hotkey: **[Alt] + [F4]**

Function: The command stops the programming system. If a project that has been modified since the last time it was saved is currently open, a dialog appears asking whether you want to save the project.

Call: Menu **File**



17.1.18 Command Page Setup...

Symbol: 

Function: The command opens the **Page settings** dialog for configuring the layout for the print version of the project content.


Call: Menu **File**

Requirement: An editor window is active.

See also:

- [Command Print \[▶ 872\]](#)

17.1.19 Command Print

Symbol: 

Function: The command opens the standard Windows dialog for printing documents.

Call: Menu **File**

Requirement: An editor window is active.

17.2 Edit

17.2.1 Standard Commands

TwinCAT offers the following standard commands:

- Undo



, hotkeys: **[Ctrl] + [Z]**

- Redo:



, hotkeys: **[Ctrl] + [Y]**

- Cut



, hotkeys: **[Ctrl] + [X]**

- Copy:



, hotkeys: **[Ctrl] + [C]**

- Paste:



, hotkeys: **[Ctrl] + [V]**

- Delete:



, hotkeys: **[Del]**

- Select All: Hotkey: **[Ctrl] + [A]**

Call: **Edit** menu, PLC project tree context menu, Editor window context menu

Not all editors support the **Paste** command, or its application may be limited. In graphical editors, the command is only supported if the paste operation creates a correct construct.

In the PLC project tree, the command refers to the currently selected object. Multiple selections are possible.

17.2.2 Command Delete

Hotkey: **[Del]**

Function: The command removes the selected PLC object from the solution. The object is retained in the project directory.

Call: PLC object context menu

17.2.3 Command Select All

Hotkey: **[Ctrl + A]**

Function: The command selects the entire content.

Call: **Edit** menu, Editor window context menu

17.2.4 Command Input Assistant

Symbol:

Hotkey: **[F2]**

Function: The command opens the **Input Assistant** dialog, which provides support for selecting a programming element, depending on the current cursor position.

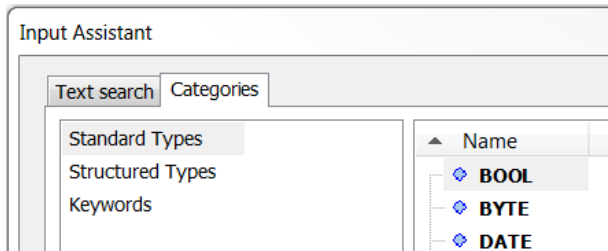
Call: **Edit** menu, Editor window context menu

Requirement: A POU is open in the editor, and the cursor is in a program line.

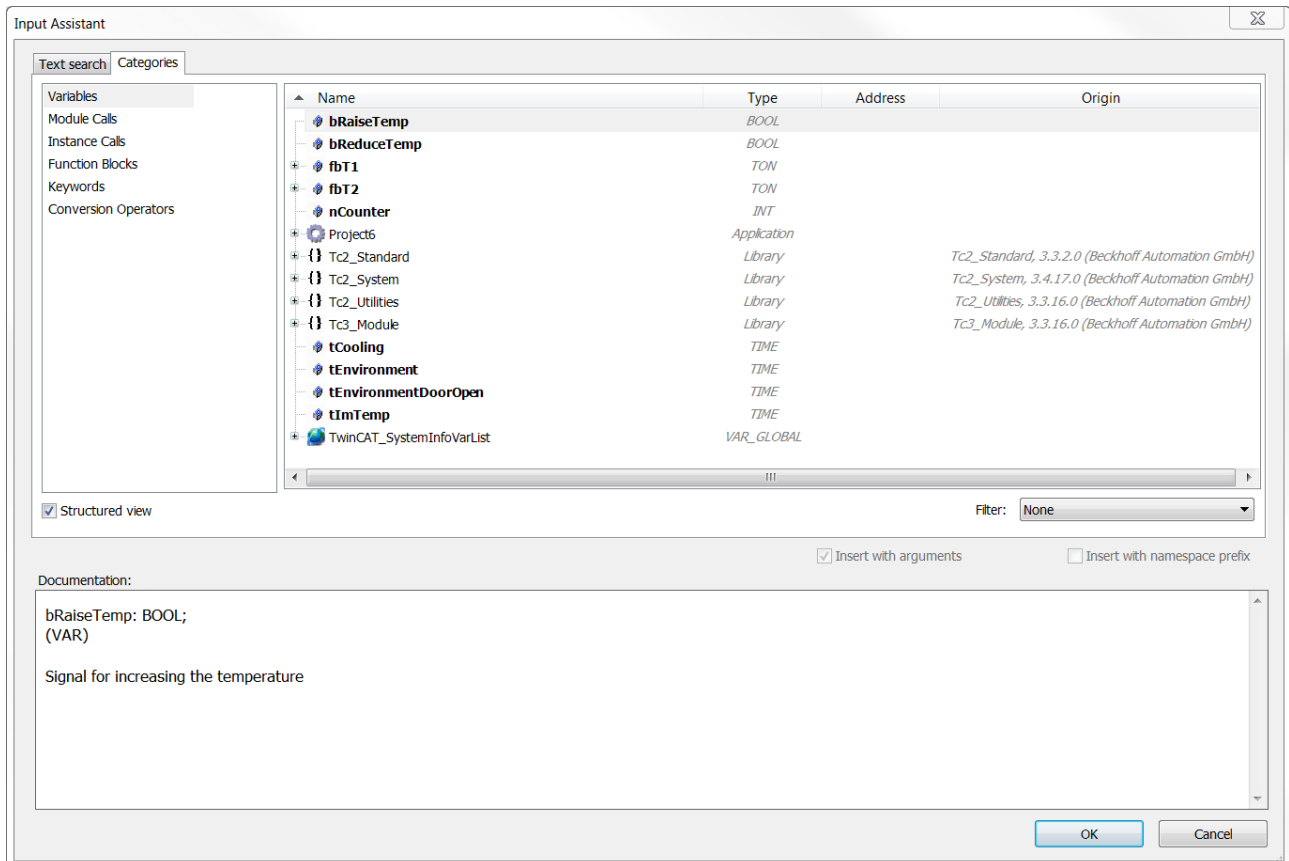
Input Assistant dialog - Categories tab

The dialog offers all programming elements, which you can add in the editor at the current cursor position. The elements are sorted by categories. You can additionally set a filter for the scope in the **Variables** category, such as Local Variables, Global Variables or Constants.

Detail of the **Input Assistant** dialog in the declaration part of the editor:



Input Assistant dialog in the implementation part of the editor:



Structured view	<input checked="" type="checkbox"/> : The elements are displayed in a structure tree. You can hide or show the column Type, Address and Origin by right-clicking on the column title in a submenu. <input type="checkbox"/> : The elements are shown in a flat structure.
Filter	You can set an additional filter for the variable type in the drop-down list box.
Display documentation	<input checked="" type="checkbox"/> : A description of the selected item is displayed.
Insert with arguments	<input checked="" type="checkbox"/> :TwinCAT inserts elements that have arguments, such as functions, at the cursor position with these arguments. Example: If you insert the function block fb1, which contains an input variable fb1_in and an output variable fb1_out, "with arguments", it looks as follows in the editor: fb1(fb1_in:= , fb1_out=>)
Insert with namespace prefix	<input checked="" type="checkbox"/> : TwinCAT inserts the selected element with the namespace prefixed. In the case of library function blocks, you cannot use the check box if the properties of the library specify that the namespace is mandatory.

Input Assistant dialog - Text search tab

You can use the tab to search for specific objects. When you type one or more characters into the search box, the results window lists the names of all objects whose name contains this search string. Double-click the desired object to insert it in the editor at the current cursor position.

Filter	Restrict the search to a specific variable category.
--------	--

See also:

- PLC documentation: [Using the input wizard](#) [▶ 135]

17.2.5 Command Auto Declare


Function: The command opens the **Auto Declare** dialog, which supports the declaration of a variable.

Call: Edit menu; Editor window context menu

Requirement: A POU is open in the editor and the cursor is in a program line.

The auto-declaration function opens the **Auto Declare** dialog even if the cursor is in the implementation part of a POU in a line that contains the name of an undeclared variable. To do this, you must have activated the option **Declare unknown variables automatically (AutoDeclare)** in the TwinCAT options (**Extras > Options > TwinCAT > PLC Programming Environment > Smart Coding**).

Due to the smart tag feature, the command **Auto Declare** also appears if you place the cursor on a variable

that has not been declared in the implementation part of the ST editor and then click  (available from build 4026).

Auto Declare dialog

The screenshot shows the 'Auto Declare' dialog box. It features a title bar with the text 'Auto Declare' and a close button (X) in the top right corner. The main area is divided into several sections:

- Scope:** A dropdown menu showing 'VAR'.
- Name:** A text input field containing 'nVar1'.
- Type:** A dropdown menu showing 'INT' with a right-pointing arrow button.
- Object:** A dropdown menu showing 'MAIN [SamplePLCProject]'.
- Initialization:** An empty text input field with an ellipsis button (...).
- Address:** An empty text input field.
- Flags:** Three checkboxes: 'CONSTANT', 'RETAIN', and 'PERSISTENT', all of which are currently unchecked.
- Comment:** A large empty text area with vertical scrollbars.

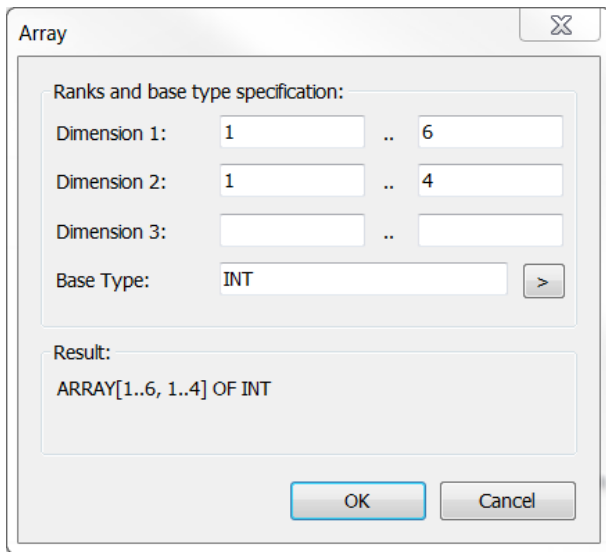
At the bottom right of the dialog, there are two buttons: 'OK' (highlighted in blue) and 'Cancel'.

Scope	Validity range of the variables that have not yet been declared. Example: VAR (default setting for local variable)
Name	Variable name not yet declared Example: bVar
Data type	<p>▼ : Lists the standard data types.</p> <p>▶ :</p> <ul style="list-style-type: none"> • Input Assistant: Opens the Input Assistant dialog • Array assistant: Opens the Array dialog <p>Example: BOOL</p>
Object	Object in which the new variable is declared. By default, this is the object you are editing. ▼ : Lists the objects in which the variable can be declared. If no objects are available for the selected scope, the entry <Create object> appears. If you select the entry <Create object>, the dialog Add object opens, which can be used to create a suitable object.
Initialization value	If you do not enter an initialization value, the variable is initialized automatically. ☐ : Opens the Initialization dialog. This approach is helpful for initializing structured variables. Example: FALSE
Address	Memory address (see PLC documentation: Addresses [▶ 754]) Example: %IX1.0
Flags	Attribute keywords <ul style="list-style-type: none"> • CONSTANT: Keyword for a constant • RETAIN: Keyword for a remanent variable • PERSISTENT: Keyword for a persistent variable (stricter than RETAIN) The selected attribute keyword is added to the variable declaration.
Comment	The tabular declaration editor displays the comment that was entered in the Comment column. In the textual declaration editor, it is displayed above the variable declaration. Example: New variable
Apply changes using refactoring	This option appears for the following validity ranges: <ul style="list-style-type: none"> • Input variable (VAR_INPUT) • Output variable (VAR_OUTPUT) • Input and output variable (VAR_IN_OUT) This option is automatically enabled if the autodeclaration options On renaming variables and On adding or removing variables, or on changing the scope are enabled in the TwinCAT options (Tools > Options > TwinCAT > PLC Environment > Refactoring) (see Dialog Options - Refactoring [▶ 979]). If this option is enabled, the variable is not yet declared when the dialog is closed. Instead, the Refactoring dialog opens, in which you can edit the changes further.
OK	The variable is declared and appears in the declaration. Example: <pre>VAR // New variable bVar: BOOL := FALSE; END_VAR</pre>

See also:

- PLC documentation: [Declaring variables](#) [▶ 66]

Array dialog

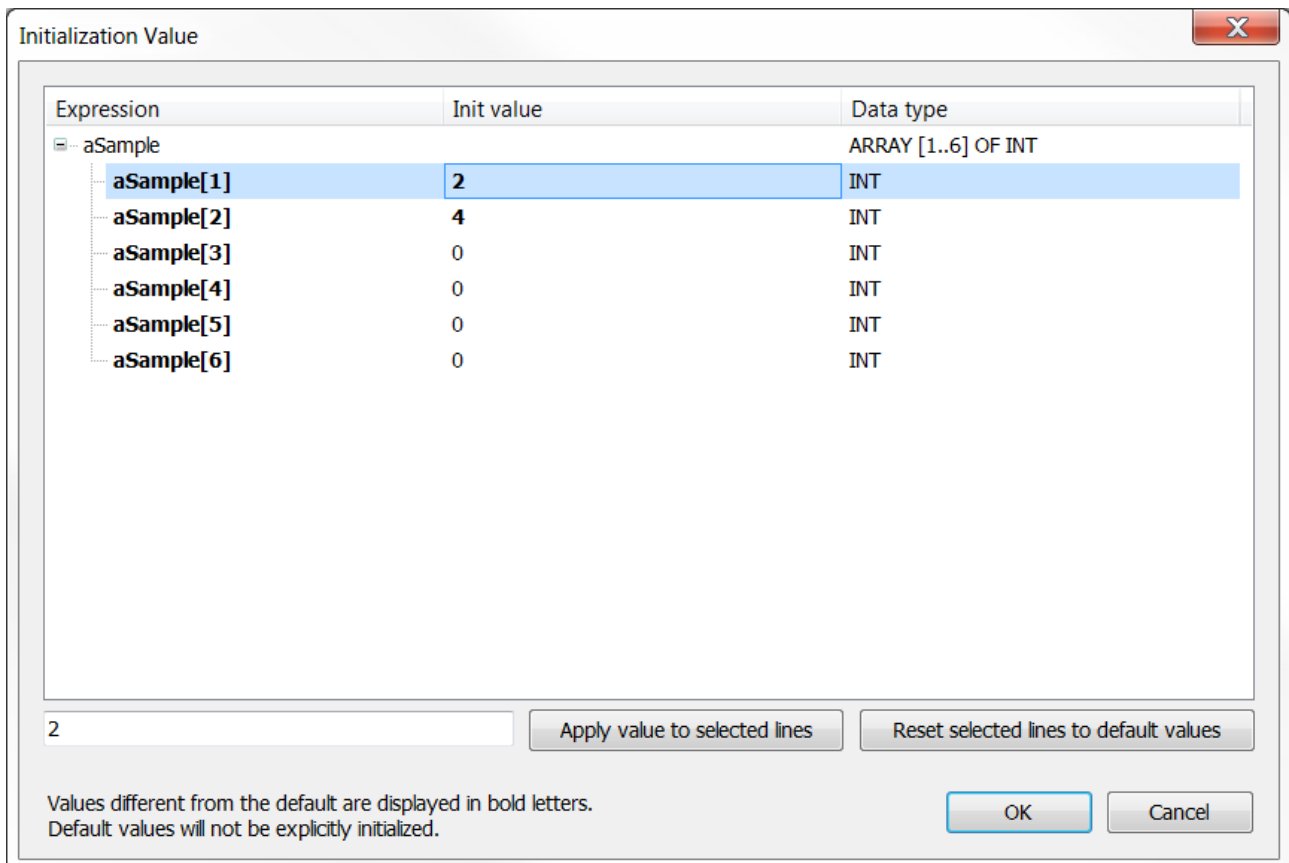


Ranks and base type specification	Definition of the field sizes (dimension) by entering the lower and upper bounds and the base type of the array. You can enter the base type directly or by using the Input Assistant or Array dialog, if you click the button.
Result	Display of the defined array.



TwinCAT only reinitializes variables if you have changed the initialization values of the variables.

Initialization Value dialog



Listing of the variables with name (expression), initialization value and data type. Changed initialization values are shown in bold.	
Input field below the list	Enter an initialization value for the selected variable(s).
Apply value to selected lines	Change the initialization value of the selected line(s) according to the value of the input field.
Reset selected lines to default values	Setting the default initialization values.
OK	TwinCAT adopts the initialization values in the Auto Declare dialog.

If the variable to be initialized via this dialog is a function block instance with extended FB_Init method, another table is displayed above the Initialization value table (see PLC documentation: [Methods FB_init, FB_reinit and FB_exit \[► 848\]](#)). This table lists the additional FB_Init parameters. The meaning and operation is basically the same as the table below with the following differences:

- All variables must be assigned initialization values. Otherwise OK is not selectable.
- For complex data types (structures, arrays) no components contained in them are displayed (type cannot be expanded). In this case, the complex type must be initialized with a corresponding variable.

For FB_Init parameters configured in this way, a corresponding symbol is displayed behind the initialization value in the **Auto Declare** dialog.

See also:

- PLC documentation: [Using the input wizard \[► 135\]](#)

17.2.6 Command Add to Watch

Symbol: 

Function: The command adds the variable on which the cursor is currently positioned to a Watch List for online monitoring.

Call: Context menu


Requirement: The PLC project is in online mode and the cursor is placed on a variable in an editor.

The command inserts the variable into the currently open Watch List. If no Watch List is currently open, the command inserts the variable into Watch List 1 and opens its view.

See also:

- PLC documentation: [Using Watch Lists \[► 226\]](#)
- PLC documentation: [Monitoring Values \[► 222\]](#)

17.2.7 Command Browse Call Tree

Symbol: 

Function: The command opens the **Call Tree** view, which displays the calls of the function block and its callers.

Call: Editor window context menu

Requirement: A function block is open in the editor, and the cursor is placed in a variable.

17.2.8 Command Go To

Function: Command moves the cursor to a defined line of code.

Call: Menu **Edit**

Requirement: A text editor is open, and the cursor is in a program line.

The command opens a dialog with a **Line number** input field.

17.2.9 Command Go To Definition

Symbol: 

Hotkey: **[F12]**

Function: The command shows the definition point of a variable or function.

PLC editor

Call: Editor window context menu

Requirement: A POU is open in the editor, and the cursor is on a variable or function.

PLC process image

Call: Solution Explorer context menu

Requirement: The process image (project instance) is expanded and the cursor is positioned at an allocated variable in the process image.

17.2.10 Command Go to Instance

Symbol: 

Function: The command opens the instance of a function block in a new window.

Call: Editor window context menu

Requirement: The PLC project is in online mode. A POU is open in the editor, and the cursor is positioned on the instance of a function block.

The command is not available for temporary instances or instances from compiled libraries.

17.2.11 Command Go to implementation

Symbol: 

Function: The command opens the online view of a method in a new window.


Call: Context menu Editor window

Requirement: The PLC project is in online mode. A POU is open in the editor and the cursor is on a method call.



Available from TC3.1 Build 4026

17.2.12 Command Go to reference

Symbol: 

Function: The command opens in online mode the declaration location of the variable referenced by the currently focused pointer.

Call: Context menu Editor window

Requirement: The PLC project is in online mode. A POU is open in the editor and the cursor is on a pointer. The referenced variable is located in a static memory.

If the pointer does not point exactly to the beginning of the variable, a corresponding message is output when switching to the variable declaration.



Available from TC3.1 Build 4026

17.2.13 Command Find all references

Hotkey: **[Shift+F12]**

Function: The command displays all locations where a variable is used in the **Cross Reference List** view.

Call: Context menu

Requirement: A POU is open in the editor and the cursor is in a variable, or the **Cross Reference List** view is open and a variable is specified in the **Name** field.

See also:

- PLC documentation: [Find locations where the cross reference list is used \[► 157\]](#)

17.2.14 Command Navigate To

Function: The command opens a dialog for selecting a specific element to be opened.

Call: Menu **Edit**

17.2.15 Command Make Uppercase

Function: The command converts all lower-case letters in the selected code to upper-case letters.

Call: Menu **Edit > Advanced**

Requirement: A POU is open in the editor, and code is selected.

17.2.16 Command Make Lowercase

Function: The command converts all upper-case letters in the selected code to lower-case letters.

Call: Menu **Edit > Advanced**

Requirement: A POU is open in the editor, and code is selected.

17.2.17 Command View white spaces

Symbol: **a·b**

Function: The command causes control characters to be displayed for spaces and tabs.

Call: Menu **Edit > Advanced**

Requirement: A POU is open in the editor.

TwinCAT visualizes spaces by a dot and tabulators by an arrow.

17.2.18 Command Comment Selection

Hotkey: [Ctrl+K] + [Ctrl+C]

Function: The command comments out a selected code section. The code section is excluded from the compilation and has no influence on the program execution.

Call: Menu **Edit > Advanced**

Requirement: A function block is open in the editor, and code is selected.

This command can be used to create comments for the documentation of a program or program section, or to temporarily exclude a code section from compilation. The command [Command Uncomment Selection \[▶ 882\]](#) can be used to cancel a comment, so that a commented-out code section is reintegrated into the program execution.

17.2.19 Command Uncomment Selection


Hotkey: [Ctrl+K] + [Ctrl+U]

Function: The command cancels a comment and reintegrates a commented-out code section back into the program execution.

Call: Menu **Edit > Advanced**

Requirement: A function block is open in the editor and code is selected which was previously commented out using the command [Command Comment Selection \[▶ 882\]](#).

17.2.20 Command Quick Find

Symbol: 

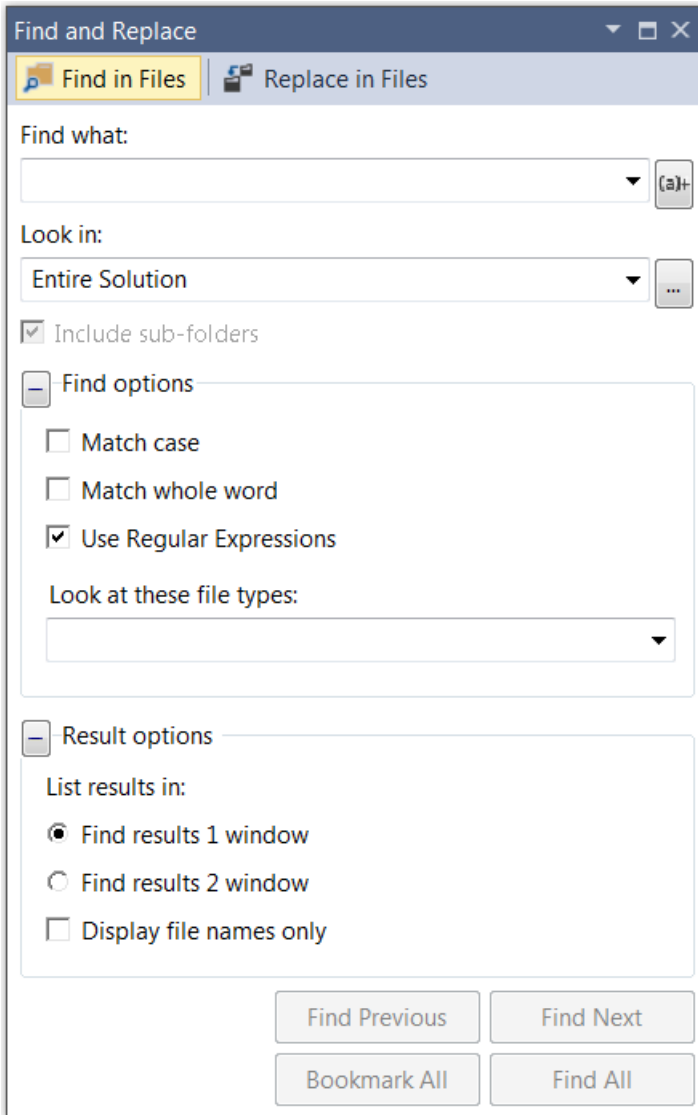
Hotkey: [Ctrl] + [F]







Function: The command scans the project or parts of it for a specific string.

Call: Menu **Edit > Find and Replace**

The command opens the **Find and Replace** dialog (**Find in Files** button is active), in which you enter the search string and the search options.

Find and Replace dialog




Replace in Files	Switches to the Find and Replace dialog (button Replace in Files is active)
Find what	Search string.
Look in	 : Selection list with the objects that are searched: Entire solution: All editable places in all objects of the project are scanned. Current project: All open documents: All editors that are currently open in a window are scanned. Current document: Only the editor in which the cursor is currently located is searched.  : Opens a dialog in which the objects to be searched can be defined more precisely.
Match case	 : The search is case-sensitive.
Match whole word	 : Only strings that exactly match the search string are found.
Look at these file types	Drop-down list for selecting a file type
Use Regular Expressions	Enables the  button, which will help you enter regular expressions. This function is not supported for PLC editors!
Display file names only	 : Only file names are displayed.
Find Next	Start the search. The next search result is displayed at its position in the corresponding editor.
Find All	All search results are shown in the message window. The object and the exact position of the search result are displayed. <ul style="list-style-type: none"> • (Decl): Declaration part of the object • (Impl): Implementation part of the object Double-click on the list entry to display the search result in the editor.

See also:

- [Command Quick Replace \[► 884\]](#)
- PLC documentation: [Find and Replace in the entire project \[► 159\]](#)

17.2.21 Command Quick Replace

Symbol: 

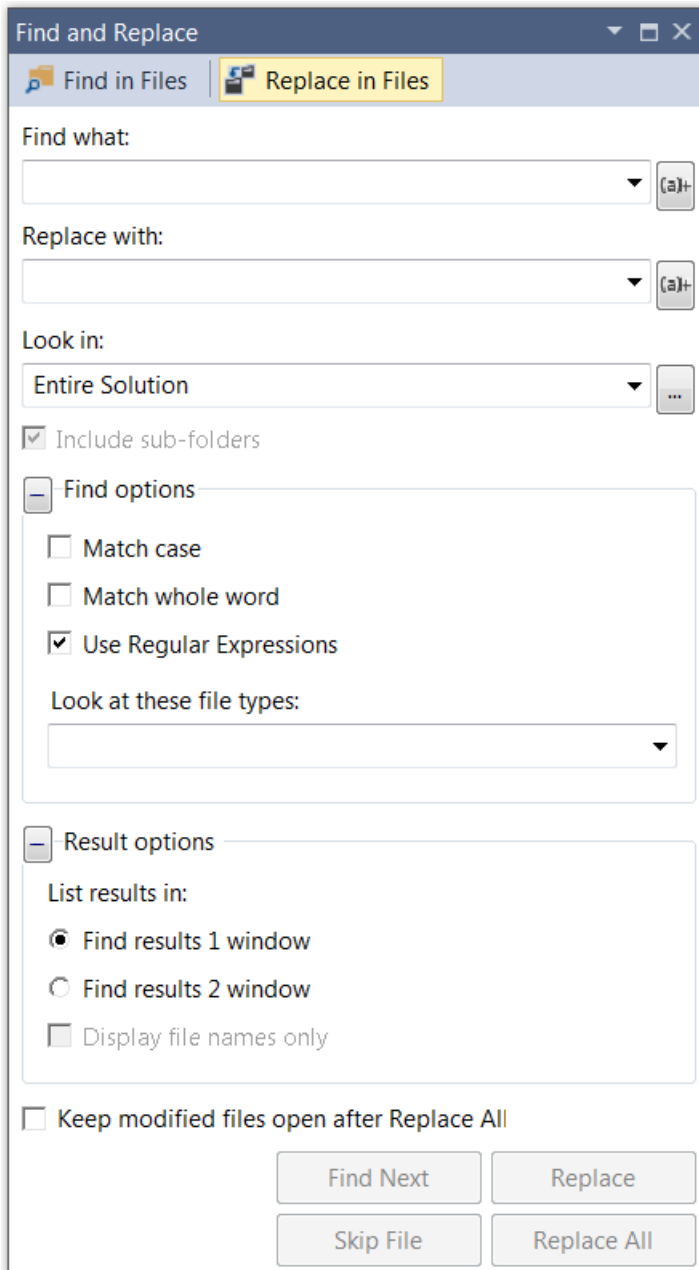
Hotkey: **[Ctrl] + [H]**

Function: The command scans the project or parts of it for a specific string and replaces it.

Call: Menu **Edit > Find and Replace**

The command opens the **Find and Replace** dialog (**Replace in Files** button is active), in which you enter the string to be replaced and the new string, along with the search options.

Find and Replace dialog



In addition to the options in the "Find" dialog, the following settings are possible:

Replace with	Input field for the new string.
Replace	The next string that is found is highlighted in the editor and replaced (step-by-step replacement).
Replace All	All strings that are found are replaced at once, without being displayed in the editors.
Keep modified files open after Replace All	The editors of the found objects remain open.

See also:

- [Command Quick Find \[► 882\]](#)
- [PLC documentation: Find and Replace in the entire project \[► 159\]](#)

17.2.22 Command Switch write mode

Hotkey: **[Insert]**

Function: This command activates overwrite mode or insert mode.

Call: Double-click **[INS]** or **[OVR]** in the status and information bar

Requirement: An editor window is active.

If overwrite mode is active, characters following the cursor are overwritten when new characters are entered. If insert mode is active, characters are inserted, and existing characters following the cursor are retained.

17.2.23 Command Rename

Function: The command allows you to rename a PLC object in **Solution Explorer**.

Call: PLC object context menu

17.2.24 Command Edit object (offline)

Function: The command opens the object offline in its editor.

Call: **Project** menu, context menu

Requirement: The PLC project is in online mode. An object is selected in the PLC project tree.

This means that you can also edit the object in online mode. The change is then transferred to the controller with the command **Online Change** or **Download**.

See also:

- [Command Online Change \[► 955\]](#)
- [Command Download \[► 955\]](#)

17.2.25 Command Rename '<variable>'

Function: The command opens the **Rename** dialog for renaming an object or a variable.

Call: Context menu PLC object, context menu Editor window > Refactoring

Requirement: An object is selected in the PLC project tree, or the cursor is positioned before or on a variable identifier in the declaration part of a programming object.

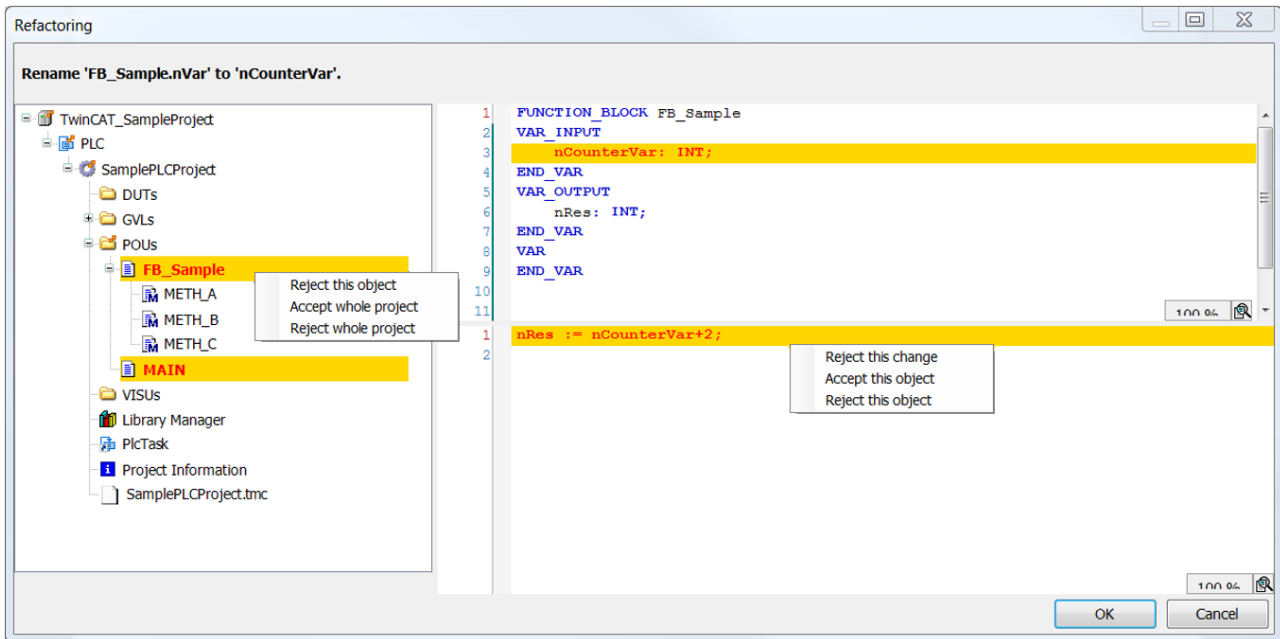
You can rename:

- Variables
- POU's
- GVL's
- Methods
- Properties

Rename dialog

Current name	Name of the object or variable
New name	Input field for a new name. If the name that is entered already exists, TwinCAT reports this directly under this input field.
OK	Can be activated if you have entered a valid name in New Name . Opens the Refactoring dialog. The respective objects and locations are color-coded in both windows. In both windows, you can specify an action for each location. Various commands are available in the context menu.

Refactoring dialog



The dialog shows all the usage locations within the project. The respective objects and locations are highlighted in color.

Left dialog part	Navigation tree of the project with the respective object.
Right part of the dialog	Displays the position within an object where the current name occurs.
In both windows, you can specify an action for each location. The commands described below are available in the context menu.	
Reject this change	Discarding the individual change in the right part of the dialog.
Accept this object	Accept all changes in the affected object
Reject this object	Discard all changes in the affected object
Accept the entire project	Accept all changes in the project
Reject the entire project	Discard all changes in the project

TwinCAT shows accepted changes with a yellow background, discarded changes with a gray background.

See also:

- PLC documentation: [Refactoring](#) [▶ 160]

17.2.26 Command Add '<variable>'

Symbol:

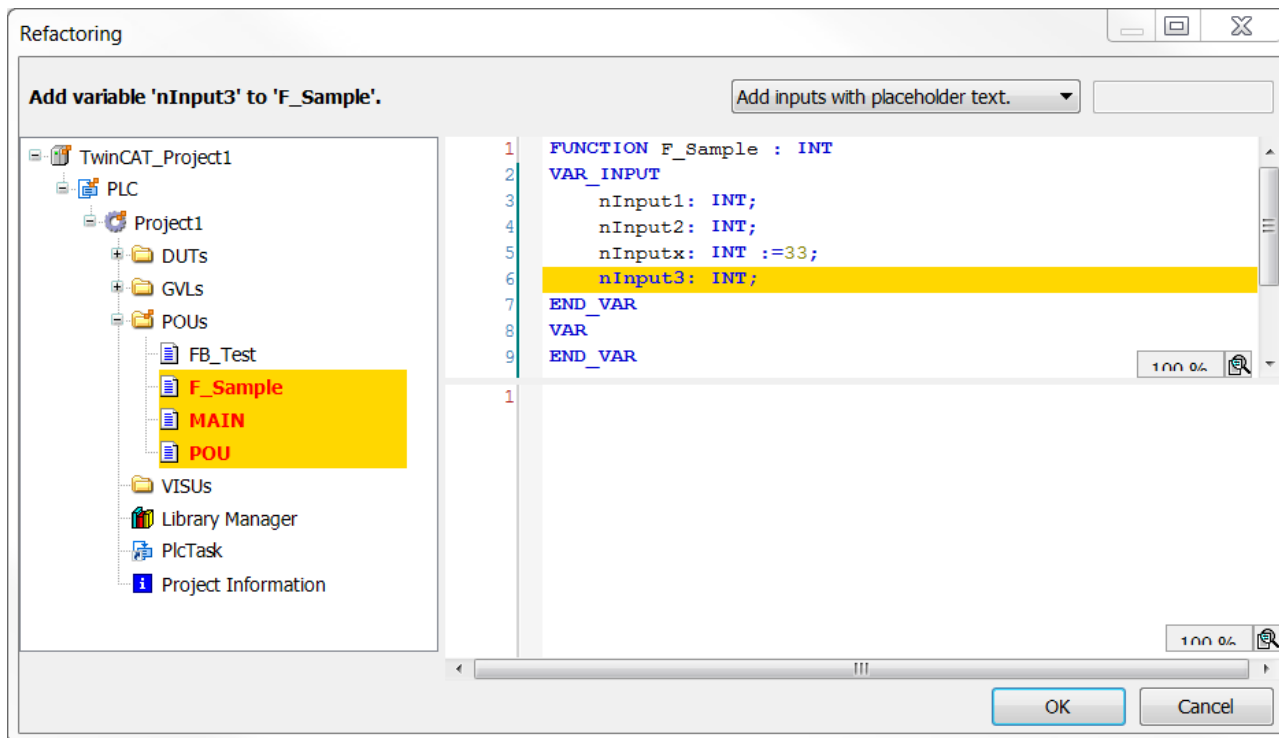
Function: This command allows you to declare a new variable in a POU and automatically update it at the point of use of the POU.

Call: Context menu Editor window > Refactoring

Requirements: The focus is on the declaration part of a POU.

The command opens the default dialog for declaring the variable. After closing the declaration dialog with **OK** the two-part **Refactoring** dialog appears.

Refactoring dialog



<p>Left dialog part</p>	<p>Navigation tree of the project. Color-coding of the function blocks in which the POU is used: Red font with yellow background. After clicking on the POU object, the detailed view opens in the right part of the dialog.</p>
<p>Right part of the dialog</p>	<p>Declaration part and implementation of the POU in whose declaration the variable is added. Color-coding of the change points: Newly added declaration in blue font and yellow background.</p>

Before you decide which changes you want to apply at which points, select the desired option from the drop-down list at the top right:

<p>Add inputs with placeholder text</p>	<p>Default placeholder text: <code>_REFACTOR_</code>; editable The placeholder text defined here appears at the usage points of the newly added variables in the implementation code. It is used to find the affected locations.</p>
<p>Add inputs with the following value</p>	<p>Initialization value for the new variable</p>

Commands for accepting or rejecting the change(s) are available in the context menu of the change point(s), both in the left and right part of the dialog. See also the description of the Command Rename '<variable>' [[▶ 886](#)].

Examples:

1. The function F_Sample is assigned a new input variable “nInput3” with initialization value “1” via refactoring. The change has the following effect:

Before:

```
F_Sample(nVarA + nVarB, 3, TRUE);
F_Sample(nInput1:= nVarA + nVarB, nInput2 :=3 , nInputx := TRUE);
```

After:

```
F_Sample(nVarA + nVarB, 3, TRUE, nInput3 := 1);
F_Sample(nInput1:= nVarA + nVarB, nInput2 :=3 , nInputx := TRUE, nInput3 := 1);
```

2. The function F_Sample is assigned a new input variable “nInput3” with placeholder text “_REFACTOR_” via refactoring:

Before:

```
F_Sample(nInput1 := nVarA + nVarB, nInput2 := 3, nInputx := TRUE);
F_Sample(nVarA + nVarB, 3, TRUE);
```


After:

```
F_Sample(nInput1 := nVarA + nVarB, nInput2 := 3, nInputx := TRUE, nInput3 := _REFACTOR_);
F_Sample(nVarA + nVarB, 3, TRUE, nInput3 := _REFACTOR_);
```

See also:

- PLC documentation: [Refactoring \[▶ 160\]](#)
- PLC documentation: [Auto Declare dialog \[▶ 876\]](#)

17.2.27 Command Remove '<variable>'

Symbol: 

Function: The command removes an input or output variable from the POU and all POU usage points.

Call: Context menu Editor window > Refactoring

Requirements: The cursor is on the identifier of the variable to be removed in the declaration part of the POU.

The command first opens a dialog showing the information about the desired removal. After confirmation, the **Refactoring** dialog appears. A description of the dialog can be found in section “[Command Add '<variable>' \[▶ 887\]](#)”.

If you accept the changes in **Refactoring** dialog, the corresponding input or output parameters are deleted at the usage points of the affected POU.



In CFC, only the connection of the removed input or output to the function block is removed. The input or output itself is retained in the chart.

Sample in ST:

You use **Refactoring** to remove the input variable “nInput4” in a POU. An automatic adjustment is made at the respective usage points:

Before the removal:

```
F_Sample(nInput1 := nVarA + nVarB, nInput2 := 3, nInput4 := 1, nInput5 := TRUE);
F_Sample(nVarA + nVarB, 3, 1, TRUE);
```


After the removal:

```
F_Sample(nInput1 := nVarA + nVarB, nInput2 := 3, nInput5 := TRUE);
F_Sample(nVarA + nVarB, 3, TRUE);
```

See also:

- PLC documentation: [Refactoring \[▶ 160\]](#)

17.2.28 Command Reorder variables

Symbol: 

Function: In the declaration editor, this command enables you to change the sequence of the variables in the currently focused scope VAR_INPUT, VAR_OUTPUT or VAR_IN_OUT.

Call: Context menu of the currently focused scope in the declaration editor

Requirement: The focus is in the declaration of one of the above scopes, and more than one variable is declared in it.


The command opens the **Rearrange** dialog with a list of all declarations of the currently focused scope. You can use the mouse to drag a selected declaration up or down to another position.

See also:

- PLC documentation: [Rearranging variables in the declaration \[► 162\]](#)

17.3 View

17.3.1 Command Open object

Symbol: 

Function: The command opens the object in its editor.

Call: **View** menu, context menu PLC object, double-click the PLC object

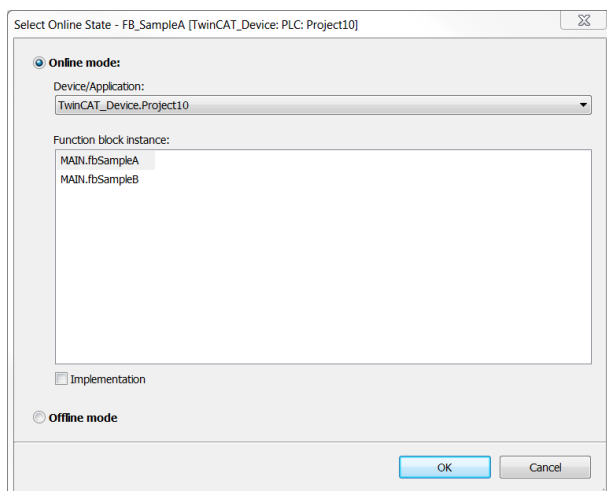
Requirement: An object is selected in the PLC project tree.

In online mode, the dialog **Select Online State** opens, in which you can choose the view in which view the object should be opened. The dialog does not open when the object selection is unambiguous. In this case, the object is opened directly in online mode.

Select Online State dialog


Function: The dialog determines how an object (function block, etc.) that was not yet open in offline mode is to be opened in online mode. You can select whether to open an instance or the basic implementation of the object itself (optionally in online or offline mode).

Requirement: The PLC project contains several instances of the selected object.



Online Mode	Activate the option to obtain a view in online mode.
Device/Application	Shows the application (project) to which the object is assigned.
Function block instance	If the object is a function block, a list of all instances currently used in the application appears.
Implementation	Select this option to display the basic implementation of the function block, regardless of the selected instance. This option has no function for non-instantiated objects.
Offline mode	Activate the option to obtain a view in offline mode.

17.3.2 Command Textual view

Symbol: 

Function: The command opens the declaration editor in text view.

Call: Button at the right edge of the editor



```

1 PROGRAM MAIN
2 VAR
3     nVarA : INT;
4     nVarB : INT;
5 END_VAR
6

```

See also:

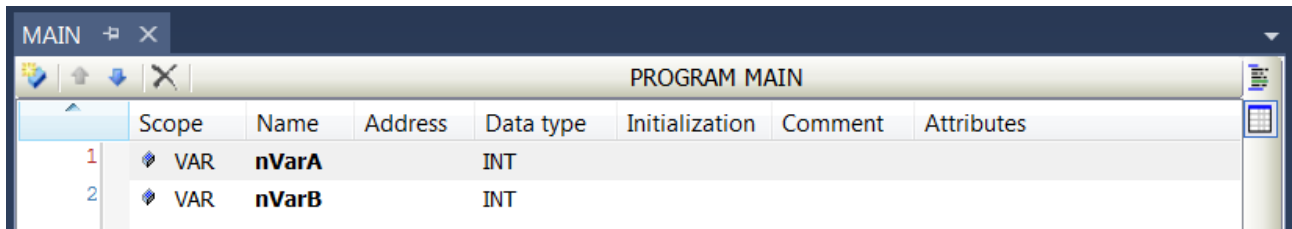
- PLC documentation: [Using the Declaration Editor \[► 70\]](#)

17.3.3 Command Tabular view

Symbol: 

Function: The command opens the declaration editor in tabular view.

Call: Button at the right edge of the editor




	Scope	Name	Address	Data type	Initialization	Comment	Attributes
1	VAR	nVarA		INT			
2	VAR	nVarB		INT			

See also:

- PLC documentation: [Using the Declaration Editor \[► 70\]](#)

17.3.4 Command Full screen

Symbol: 

Hotkey: [Ctrl] + [Shift] + [F12]

Function: The command switches the TwinCAT display to full-screen mode.

Call: View menu

When you enable the command, the main window of the TwinCAT user interface is displayed in full screen mode. You can return to the preset size by disabling the command again.

17.3.5 Command Toolbars

Function: The command opens a menu for selecting the displayed toolbars.


Call: View menu, Toolbar area context menu

In the menu that opens, select the toolbars you want to show or hide. The command works as an option, i. e. when a toolbar is displayed, it appears in the menu with a check mark in front of it.

See also:

- TC3 User Interface documentation: Customizing Toolbars

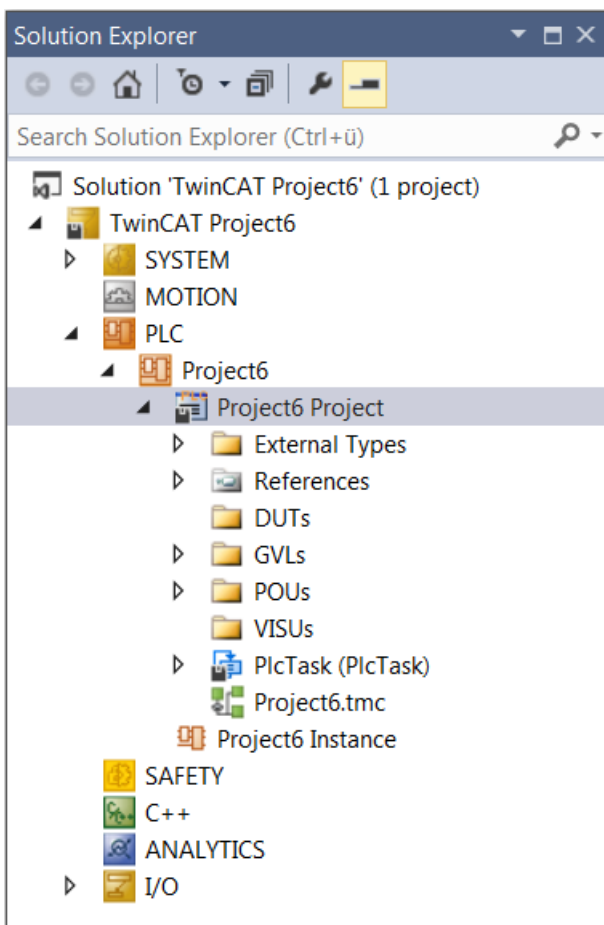
17.3.6 Command Solution Explorer

Symbol: 


Function: The command opens the **Solution Explorer** view.

Solution Explorer view

The **Solution Explorer** view shows the TwinCAT 3 project with the corresponding project elements in a structured form. In this view, you can open objects for editing and configuring.



17.3.7 Command Properties Window

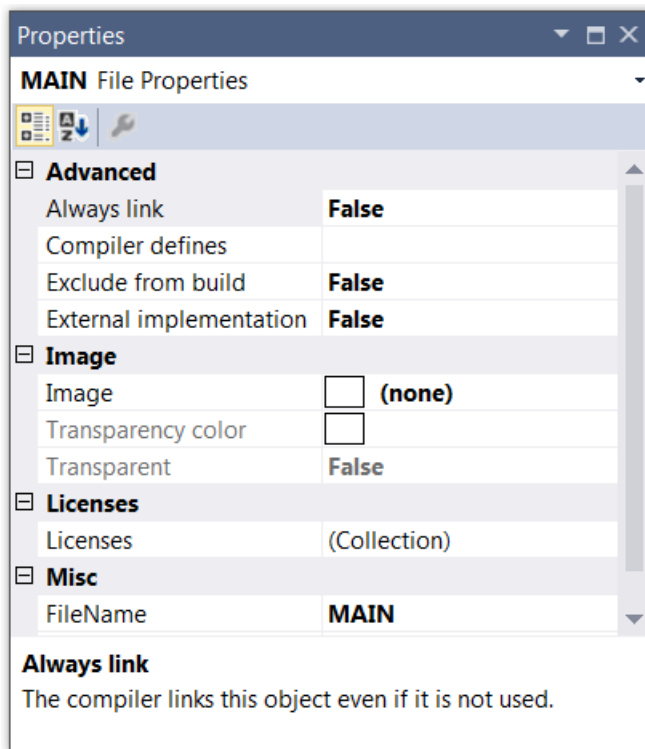
Symbol: 

Function: The command opens the **Properties** view.

Call: View menu


Properties view

The **Properties** view shows the properties for the object currently selected in the Solution Explorer. By default, the element properties are sorted by category in a table. Clicking on the plus or minus sign in front of the category to show or hide the associated parameters. A mouse click on the value field of a parameter activates the input mode, in which you can edit the value or property. You can filter or sort the Properties view.



You can also call the Properties window from the context menu of an object in the PLC project tree. A description of the command as well as various object properties can be found in section [Command Properties \(object\) \[► 901\]](#).

17.3.8 Command Toolbox

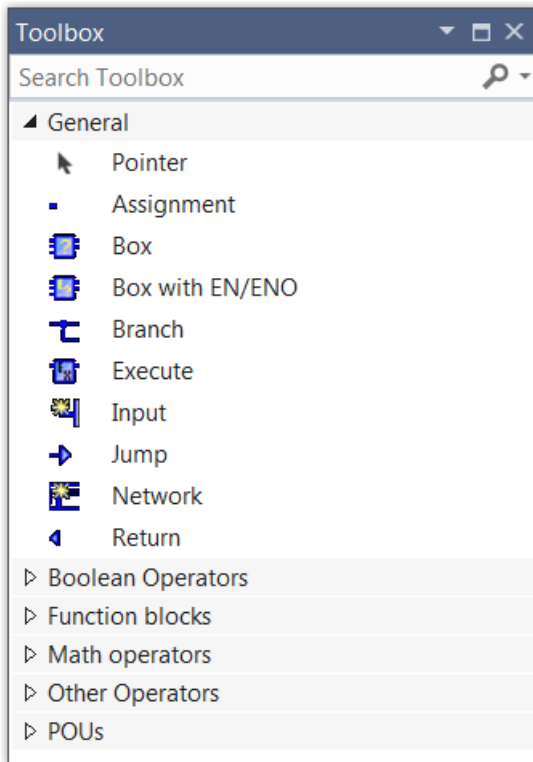
Symbol: 

Function: The command opens the **Toolbox** view.


Call: View menu

Toolbox view

The **Toolbox** view shows the existing tools for the currently active editor. This view is available by default with a graphic editor or a visualization. It contains the graphical programming elements, which you can drag-and-drop into the editor window.



17.3.9 Command Error List

Symbol: 





Function: The command opens the **Error List** view.

Call: **View** menu


Error List view

The **Error List** view displays errors, warnings and messages related to syntax checking, compile process (compile errors, code size), import processes or Library Manager. The messages are displayed in tabular form.




Filter	Drop-down list for selecting the set of code files to be used: Open Documents Current Project Current Document
Message categories	Click the message category symbol to show or hide messages. Next to each symbol, TwinCAT displays the number of messages that have occurred.
<ul style="list-style-type: none"> •  : Error •  : Warning •  : Information 	
Clear	Clears the message display
	Clears the filter settings of the error list
Severity level (message category)	Message text with the causing object and the position within the object. Double-click a message entry in the table to go to the source text position.
Code	
Description	
Project	
File	
Line	

Commands in the context menu

Clear	Clears the message display
Show columns	Adds additional columns, which may describe an error in more detail
 Copy	Copies the selected error message
Next Error	Selects the next message. The source text position of the next error is displayed.
Previous Error	Selects the previous message. The source text position of the previous error is displayed.

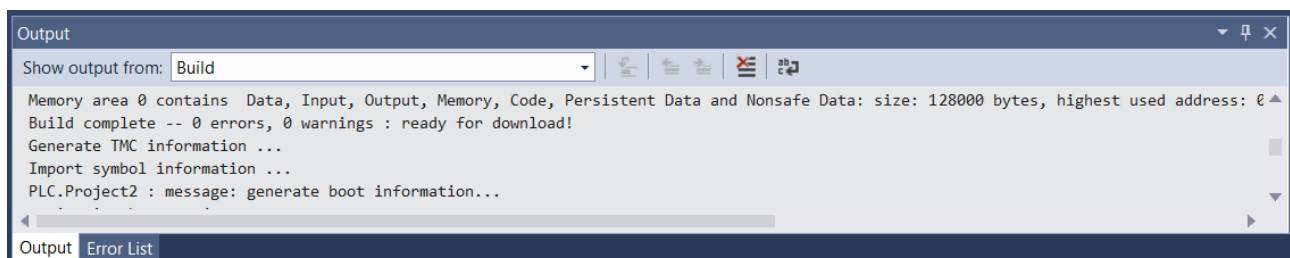
17.3.10 Command Output






Symbol: 

Function: The command opens the **Output** view.






Call: **View** menu

Output view




Message category	The messages are categorized by component or functionality and are available in a selection dialog. You can filter the message display by selecting a category.
 Find the message in the code	The source text position of the message is displayed. Requirement: A message is highlighted.
 Go to the previous message	The previous message is selected.
 Go to the next message	The next message is selected.
 Delete all	Deletes all messages
 Toggle line break	Line break is enabled or disabled

Commands in the context menu

 Copy	Message text is copied
 Delete all	Deletes all messages
 Go To Location	The source text position of the message is displayed. Requirement: A message is highlighted.
 Go to next position	The next message is selected.
 Go to previous position	The previous message is selected.

17.4 Project

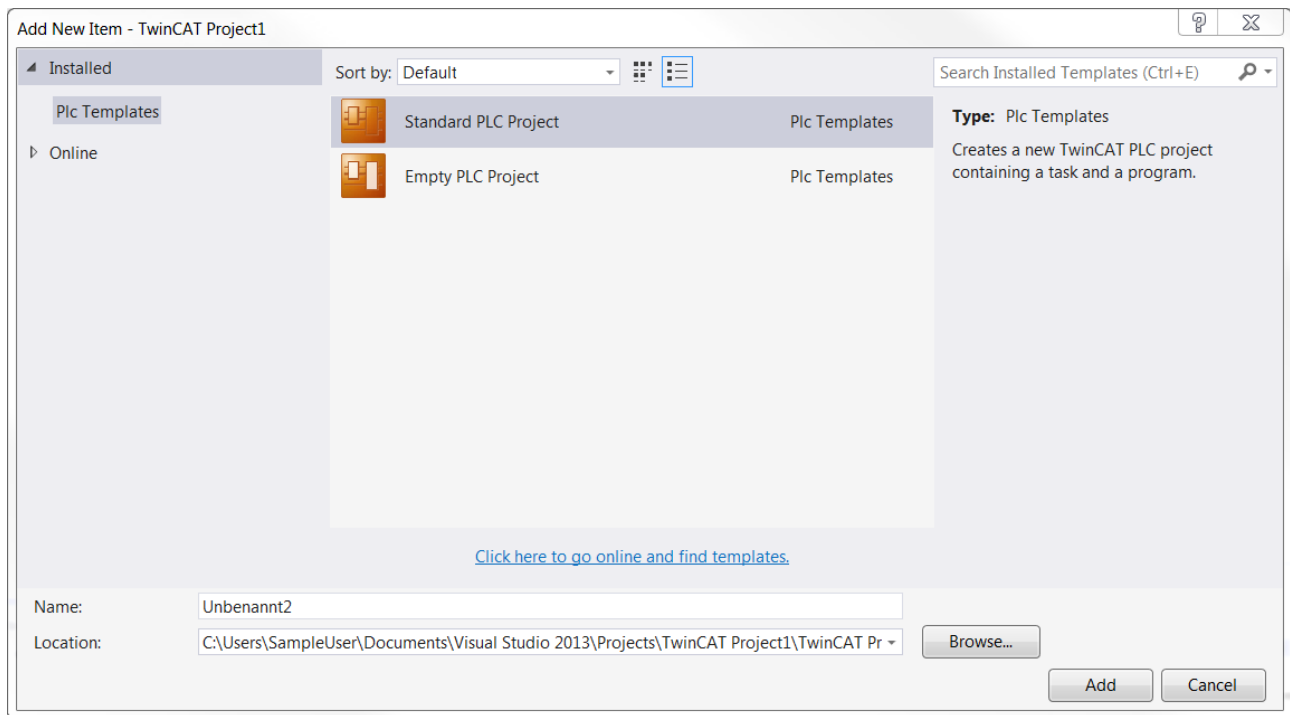
17.4.1 Command Add New Item (project)

Symbol: 

Function: The command opens the **Add New Item** dialog, through which you can create a new PLC project file. (The command is only available if a PLC node is selected.)

Call: **Project** menu or PLC object context menu in the **Solution Explorer**

Requirement: The PLC node is selected in the TwinCAT project tree.



Plc Templates	Select one of the listed templates. The template determines the basic configuration of a PLC project file. The following templates are available by default: <ul style="list-style-type: none"> • Standard PLC Project: Creates a new TwinCAT PLC project (*.project). The project is supported by a wizard and contains a Library Manager, a POU "MAIN" program and a referenced task. • Empty PLC Project: Creates an "empty" TwinCAT PLC project for library projects (*.library). The project contains no objects or devices.
Name	Define the name of the new project here. The default name depends on the selected template (usually "Unnamed<n>") and contains a sequential number to ensure that the project name is unique in the file system. You can change the default name according to the file path conventions of the local operating system. A file extension (e.g. .project) can be added. By default, the selected template automatically adds the appropriate extension.
Location	Specify the location for the new project file. The default path depends on the selected template. You can either use the Browse... button to open the default browser and specify a path, or you can use the corresponding drop-down list to select a previously entered path.
Add	Clicking Add creates a new project based on your settings. If the cursor is placed on an error symbol, a tooltip provides information on how to proceed. If another PLC project is already open, a dialog opens asking you if you want to save and close the project before the new project is opened. The name of the new project is then displayed in the title bar of the TwinCAT XAE frame window. An asterisk ("*") after the name indicates that the project has been modified since it was last saved.

See also:

- PLC documentation: [Your first TwinCAT 3 PLC project |> 24\]](#)
- PLC documentation: [Creating and configuring a PLC project |> 54\]](#)

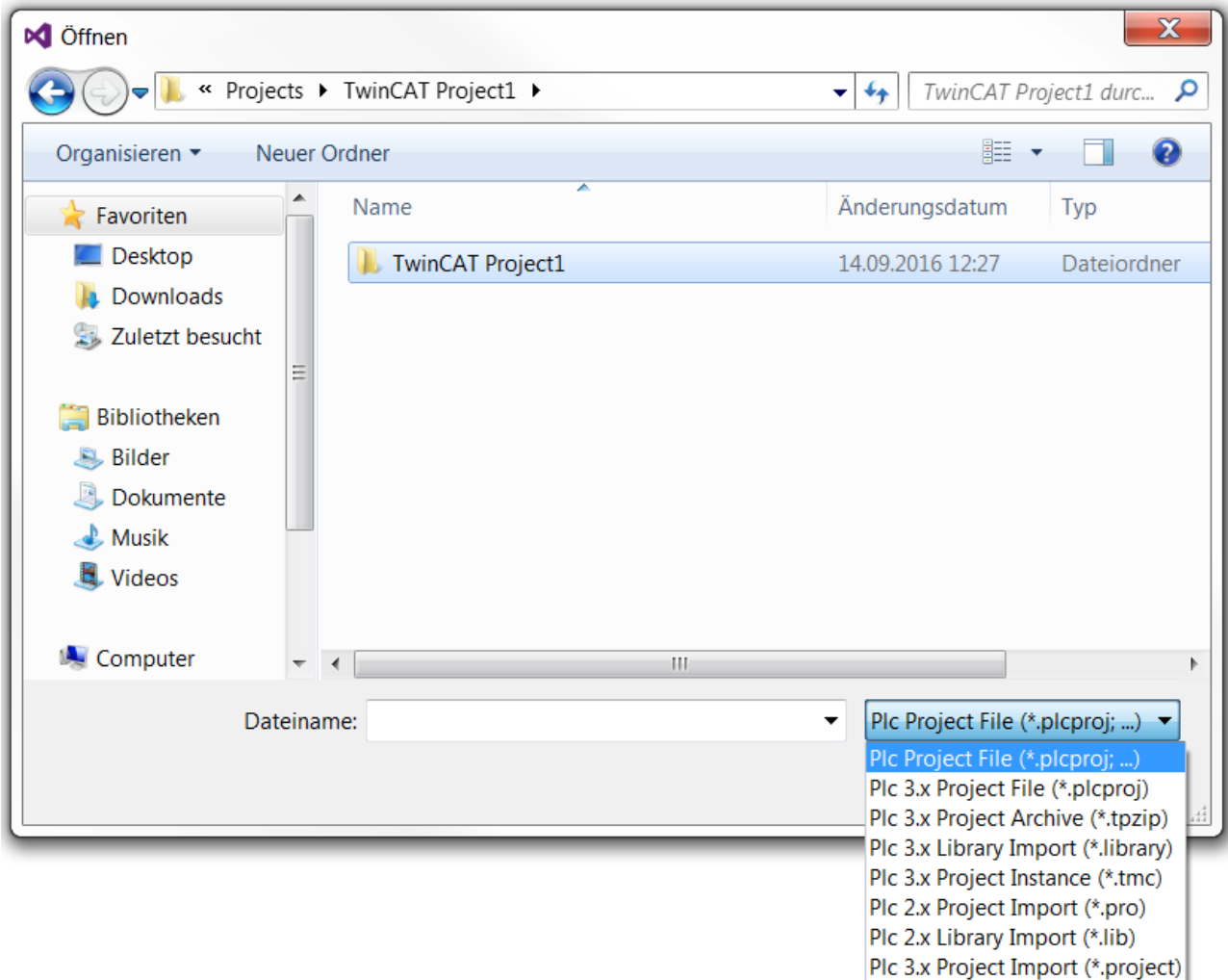
17.4.2 Command Add Existing Item (Project)

Symbol:

Function: The command opens the standard browser dialog, which can be used to search for a PLC project file and open it in the programming system. If a suitable converter is installed, you can open projects in a different format.

Call: Project menu or PLC object context menu in the **Solution Explorer**

Requirement: The PLC node is selected in the TwinCAT project tree.



File type	By default, you can set the filter to one of the following file types: <ul style="list-style-type: none"> • PLC 3.x Project file (*.PLCproject): TwinCAT 3 PLC projects with the extension ".PLCproject" • PLC 3.x Project archive (*.tpzip): TwinCAT 3 PLC project archives with the extension ".tpzip" <ul style="list-style-type: none"> ◦ See also: Command Save <PLC project name> as archive... [► 863] • PLC 3.x Library import (*.library): TwinCAT 3 PLC libraries with the extension ".library" • PLC 2.x Project file (*.pro): TwinCAT 2 PLC projects with the extension ".pro" • PLC 2.x Import library (*.lib): TwinCAT 2 PLC libraries with the extension ".lib" • PLC 3.x Project import (*.PLCproject): PLC projects with the extension ".project"
Open	The selected project file is opened or converted and then opened.

*.tpzip PLC project archive

Content of *.tpzip	The *.tpzip archive folder contains the PLC project to be archived.
Command for creating	A tpzip archive can be created with the following command: Command Save <PLC project name> as archive... [► 863]
Note on PLC projects	The files and folders stored in the archive folder for the PLC project depend on the PLC project settings of this PLC project. Settings tab [► 926]

Possible scenarios when opening a PLC project

The following scenarios are possible when you open a project:

1. [Another project is still open. \[► 899\]](#)
2. [The project was saved with an older version of TwinCAT 3. \[► 899\]](#)
3. [The project was not saved with TwinCAT 3. \[► 899\]](#)
4. [The project was not terminated properly and "Save automatically" was enabled. \[► 900\]](#)
5. [The project is read-only. \[► 900\]](#)
6. [It is a library that is installed in a library repository and retrieved from it. \[► 901\]](#)

1. Another project is still open.

You are asked if the other project should be saved and closed.

2. The project was saved with an older version of TwinCAT 3.

If the file format differs because the open project was saved with an older version of TwinCAT 3, there are two options:

- If the project cannot be saved in the format of the currently used programming system, you must update it to continue working on the project. The expression that appears at this point: **The changes you made...** refers to internal tasks of various components while the project is loaded.
- If the project can still be saved in the previous format, you can decide whether to update or retain the format. If you decide to retain the format, data loss may occur. If you decide to update the format, the project can no longer be opened with the old version of the programming system.

In addition to the file format, the versions of the explicitly inserted libraries, the visualization profile and the compiler version of the opening project may differ from those installed with the current programming system.

If newer versions are installed on the current programming system, the **Project Environment** dialog opens automatically, where you can update the versions. If no update is made at this point, this can be done later at any time via the **Options > Project Environment** dialog.

Note the compiler version

i If a project is opened that was created with an older version of the programming system and for which the latest compiler version is set in the project settings, while the project environment setting for the compiler version is set to **Do not update** in the new programming system, the compiler version that was used last in the old project continues to be used (i.e. not the "Current" version in the new environment).

3. The project was not saved with TwinCAT 3.

Case 1)

If you set the file filter when selecting the project to be opened and an appropriate converter is available, the converter is used automatically and the project is brought into the current format. The conversion is converter-specific. Usually, you are prompted to define the handling of referenced libraries or device references.

TwinCAT 3 converter

i Adaptation of a TwinCAT PLC control project to the TwinCAT 3 syntax can only be successful during import if the converter is able to compile the project without errors.

If you have set the **All Files** option when selecting the project to be opened, no converter is enabled and the **Convert Project** dialog opens. In the dialog, you need to explicitly trigger the conversion of the project by selecting one of the options.

- **Convert to the current format:** Select the converter you want to use from the selection list (application for conversion). After the conversion, the project can no longer be opened in the old version.

- **Create a new project and add a specific device:** (Not yet implemented)

● TwinCAT 2.x PLC Control project options

I The project directory path set in the TwinCAT 2.x PLC control project options and the project information are adopted in the **Project Information** dialog.

Case 2)

If libraries are integrated in the project for which "conversion mapping" has not yet been stored in the library options, the **Converting a library reference** dialog appears, in which you can define how this reference should be converted:

- **Convert and install the library:** If you select this option, the referenced library is converted to the new format and remains referenced in the project. It is automatically installed in the library repository under the **Other** category and continues to be used. If the library does not have the project information (title, version) required for an installation, you will be prompted to enter it in the **Enter Project Information** dialog.
- **Use the following library, which is already installed:** If you select the options, the referenced library is replaced by another one that is already installed on the local system. Use the **Select** button to open the **Select...** dialog. Here you can select the desired version of one of the installed libraries. This corresponds to the configuration of the version handling in the **Library Properties** dialog. An asterisk ("*") means that, as a rule, the latest version of the library available on the system is used in the project. The list of available libraries is structured in the same way as in the **Library Repository** dialog. You can sort the list by company and category.
- **Ignore the library. The reference will not appear in the converted project:** If you enable this option, the library reference is removed. The library is then no longer included in the converted project.
- **Use this mapping in future if this library is present:** If you enable this option, the settings made here in the dialog will also be applied to future project conversions, as soon as the respective library is referenced.

In the converted project, the library references are defined in the Global Library Manager in the Solution Explorer. After the conversion of the library references, the project conversion continues with the **Open Project** dialog, as described above.

For general information on library management, see section "Using libraries" in the PLC documentation.

Case 3)

When you open a TwinCAT 2.x PLC Control project that references a device (target system) for which no "conversion mapping" has yet been defined in the TwinCAT 2.x PLC Control converter options, the **Device Conversion** dialog opens, in which you can specify whether and how the old device references are to be replaced by more recent ones. The device originally used is displayed. Choose one of the following options:

- **Use the following already installed device:** Click the **Select** button to open the **Select target system** dialog, in which you can select one of the devices currently installed on the system. This device is then inserted in the **Solution Explorer** of the converted project, instead of the old one. Select the option **Select a target system...** to select one of the devices listed. The list of available devices is structured in the same way as in the **Device Repository** dialog. You can sort the list by manufacturer or by category.
- **Ignore the device. No application-specific objects will be available:** If you enable this option, no entry for the device is created in the **Solution Explorer** of the new project, i.e. the device is ignored during the conversion, and no application-specific objects, such as the task configuration, are applied.
- **Save this assignment for future reference:** If you select this option, all settings of the dialog, i.e. the displayed "conversion mapping" for the device, are saved in the TwinCAT 2.x PLC Control Converter options and applied to future conversions.

4. The project was not terminated properly and "Save automatically" was enabled.

If the **Auto Save** function was enabled in the **Load and Save** options and TwinCAT 3 PLC was not terminated regularly after the last modification of the project without saving, the **Auto Save Backup** dialog opens for handling the backup copy.

5. The project is read-only.

If the project to be opened is read-only, you are asked whether you want to open the project in write-protected mode or whether you want to unlock it.

6. It is a library that is installed in a library repository and retrieved from it.

An error message is displayed if you try and open a library project that is installed in a library repository. You cannot open a library project using this path. After closing the dialog with **OK**, the project name appears in the title bar of the user interface. An asterisk ("*") after the name indicates that the project has been modified since it was last saved.

See also:

- PLC documentation: [Open a TwinCAT 3 PLC project \[► 57\]](#)
- PLC documentation: [Open a TwinCAT 2 PLC project \[► 57\]](#)

17.4.3 Command Properties (object)

Function: The command enables the **Properties** view, which displays general information about the currently selected object.

Call: Context menu PLC object

Requirement: An object in the PLC project tree is selected.

Depending on the object currently selected, the following property areas are displayed:

- [Advanced \[► 901\]](#) (compile settings)
- [Image \[► 902\]](#)
- [Licenses \[► 902\]](#)
- [General \[► 902\]](#) (object name, object path)
- [SFC Settings \[► 905\]](#) (flags for Sequential Function Chart)
- [CFC settings \[► 905\]](#) (Execution order mode)



The special visualization properties are documented under [Visualization object \[► 402\]](#) and the special library and placeholder properties are documented under [Command Properties \[► 362\]](#).

Advanced

This area shows the settings for compiling the object.

Advanced	
Always link	False
Compiler defines	
Exclude from build	False
External implementation	False

Always bind	<p>True: The object is selected in the compiler and therefore always included in the compile information. It is therefore always compiled and loaded onto the PLC. This option becomes relevant if the object is located below an application or referenced via libraries that are also located below an application. The compile information is also used as a basis for the selectable variables of the symbol configuration.</p> <p>Alternatively, the <code>{attribute 'linkalways'}</code> [▶ 809] pragma can be used to instruct the compiler to always include an object.</p>
Compiler definitions	<p>The compiler definitions entered here are not evaluated.</p> <p>If you want to use compiler definitions, enter them in the PLC project properties. See:</p> <ul style="list-style-type: none"> • Command properties (PLC project) > Category Compile [▶ 910] • PLC documentation: Reference Programming > Pragmas > Conditional pragmas [▶ 837]
Exclude from compilation	True: The object is not included in the next compilation run.
External implementation	<p>(Late binding in the runtime system)</p> <p>The use of this functionality is only possible in special constellations. As a rule, you can ignore this option.</p> <p>True: No code is generated for this object when the project is compiled. The object is not linked until the project is loaded to the target system, provided it exists there (in the PLC runtime system or in another real-time module).</p>

Image

In this area you can assign an image to the object, which is displayed in the graphical view of the library manager and in the toolbox of the FBD/LD/IL editor. Transparency of the image can be achieved by selecting a color, which is then displayed transparently. If you select the option **Transparency Color**, you can use the rectangular button to the right of it to open the standard dialog for selecting a color.

<input type="checkbox"/> Image	
Image	<input type="text" value="(none)"/>
Transparency color	<input type="text"/>
Transparent	False

Licenses

This section contains a list of licenses for the object.

<input type="checkbox"/> Licenses	
Licenses	(Collection)

General

In this section you will find general information about the selected object.

FileName	File/object name
FullPath	Storage path/location of the object (not editable at this point)
Version	<p>File version, values:</p> <ul style="list-style-type: none"> • 1.1.0.1: This file version is used when the object is saved in XML format. • 1.2.0.0: This file version is used when the object is saved in Base64 format. Please note that objects with file version 1.2.0.0 (or higher) cannot be loaded with Engineering versions lower than TC3.1.4024! <p>(not editable at this point, indirectly configurable via the storage format, see Format property)</p>

**Engineering incompatibility of file version 1.2.0.0 (or higher) with TwinCAT 3.1 < build 4024**

Please note that objects saved with file version 1.2.0.0 (or higher) cannot be loaded with Engineering versions lower than TwinCAT 3.1.4024!

Since an object is automatically saved with file version 1.2.0.0 when using the optional Base64 format, objects with Base64 format cannot be loaded with Engineering versions lower than TwinCAT 3.1.4024.

If a PLC project contains objects with the file version 1.1.0.1 and objects with the file version 1.2.0.0, the 1.1.0.1 objects are loaded with an Engineering version lower than TwinCAT 3.1.4024. Objects with file version 1.2.0.0 are not loaded.

The file version of a file saved with file version 1.2.0.0 can be reset to 1.1.0.1 using XAE version TwinCAT 3.1.4024 or higher.

Options

In this section you will find some options that can be configured for PLC objects.

Format	<p>Individual setting option of the storage format:</p> <p>The storage format of an object can be individually configured at this point for the object types listed below.</p> <p>Storage format, values:</p> <ul style="list-style-type: none"> • XML: The object is saved in XML format. <ul style="list-style-type: none"> ◦ Objects with this storage format are stored in file version 1.1.0.1, see Version property. • Base64: The object is saved in Base64 format. <ul style="list-style-type: none"> ◦ Objects with this storage format are stored in file version 1.2.0.0, see Version property. Please note that objects with file version 1.2.0.0 (or higher) cannot be loaded with Engineering versions lower than TC3.1.4024! <p>Advantages of Base64 over XML:</p> <p>Base64 results in compressed storage, compared to XML. As a result, improved performance can be achieved with file access to these objects, which can be used, for example, when loading, moving or copying objects.</p> <p>Availability of Base64:</p> <p>The Base64 storage format is optionally available from build 4024 for the following PLC objects:</p> <ul style="list-style-type: none"> • POU's where the POU body is programmed in a graphical implementation language <ul style="list-style-type: none"> ◦ Sequential Function Chart (SFC) ◦ FBD/LD/IL (Function Block Diagram/Ladder Diagram/Instruction List) ◦ CFC (Continuous Function Chart and page-oriented CFC) ◦ UML class diagram and Statechart • POU's with a subelement (e.g. action, method) that is programmed in a graphical implementation language (for graphical languages see first key point) • Visualizations • Visualization Manager • Text lists • Recipe manager • Image pool <p>Setting option for the standard storage format:</p> <p>For a PLC project, the setting "Write object content as" in the PLC project properties (Category Advanced [▶ 920]) can be used to define the standard storage format for the object types mentioned above.</p>
Separate Linelds	<p>Value: True or False</p> <ul style="list-style-type: none"> • True: The line IDs of this POU are stored in a separate file (LineIDs.dbg). • False: The line IDs of this POU are stored in the POU itself. <p>(not editable at this point, configurable in the Write options [▶ 994])</p>
Sort	<p>Value: Name or GUID</p> <p>Specifies the sequence in which the child objects (such as methods) are stored in the parent object: either sorted by name or by GUID.</p> <p>(not editable at this point, configurable in the Write options [▶ 994])</p>
Write ProductVersion	<p>Value: True or False</p> <p>(not editable at this point, configurable via the setting "Write product version in files" in the Category Advanced [▶ 920] of the PLC project properties)</p>

SFC settings

This area displays the current settings for compiling and handling implicit variables for the SFC object currently selected.

SFC	
Use default SFC settings	True
SFC Build	
CalculateActiveTransitionOnly	False
SFC Flags	
SFCCurrentStep	Declare
SFCEnableLimit	Declare
SFCError	Declare
SFCErrorAnalyzation	Declare
SFCErrorAnalyzationTable	Declare
SFCErrorPOU	Declare
SFCErrorStep	Declare
SFCInit	Use
SFCPause	Declare
SFCQuitError	Declare
SFCReset	UseDeclare
SFCTip	Declare
SFCTipMode	Declare
SFCTrans	Declare

Use default SFC settings	True (default): With this option, the default values defined in the PLC project properties can be applied to the currently selected object and displayed in the Properties view of the object. False: This option allows you to configure SFC settings that are valid specifically for this SFC object.
Compile ("Build") and variables ("Flags")	The meaning of these items corresponds to the default settings for SFC objects, which are configured in the PLC project properties.

CFC settings



Available from TC3.1 Build 4026

This area sets the execution order mode for the selected CFC object. In the CFC editor, you freely position the elements and thus the networks. To avoid that the execution order in the CFC programming block is undefined, two modes are available.

CFC	
Explicit Execution Order	False

Explicit Execution Order	False (default): Automatic data flow mode True: Explicit execution order mode
--------------------------	--

Automatic data flow mode

In this mode, the execution order is automatically set according to data flow and, in case of ambiguity, according to network topology. The programming blocks and the outputs are numbered internally. The upper networks are executed before the lower networks and the left networks before the right networks.

Advantage: The automatically defined execution order is time and cycle optimized. You do not need any information about the internally managed execution order even during the development process.

The elements in the CFC editor are displayed without marks and without numbering. It is not possible to change the execution order manually. For networks with feedback, you can additionally set a starting point.

In the menu **CFC > Execution order** the following commands are available in this mode:

- [Command Display Execution Order \[► 1014\]](#)
- [Command Set Start of Feedback \[► 1014\]](#)

Explicit execution order mode

In this mode you can explicitly set the execution order. For this purpose, the elements are displayed in the CFC editor with marks and numbering, and menu commands are provided that allow you to determine the order.

The following commands are available in the menu **CFC > Execution order**:

- [Command Move to Beginning \[► 1015\]](#)
- [Command Move to End \[► 1015\]](#)
- [Command Forward by one \[► 1016\]](#)
- [Command Back by one \[► 1016\]](#)
- [Command Set Execution Order \[► 1016\]](#)
- [Command Order by Data Flow \[► 1017\]](#)
- [Command Order By Topology \[► 1017\]](#)



Until build 4026, this was the usual behavior of CFC programming blocks. Note that you must adjust the execution order on your own responsibility and judge the consequences and effects yourself. The execution order is constantly displayed for this purpose.

17.4.4 Command Properties (PLC project)

Symbol: 

Function: This command opens an editor window in which the properties of the project and additional project-related information can be displayed and defined.

Call: Context menu of the PLC project object (<PLC project name> Project) or **Project** menu

Requirement: A project is open.

TwinCAT stores the PLC project properties directly in the PLC project.



Scope of PLC project properties

Note that the scope differs between different project properties! Some properties affect only the PLC project whose properties you are currently configuring. Other properties, on the other hand, affect all PLC projects in the development environment. Such properties, which you can change in the project properties of a PLC project, but which also affect all other PLC projects, are titled **Solution options**.

See also:

- PLC documentation: [Configuring a PLC project \[► 60\]](#)

17.4.4.1 Category Common

The category **Common** contains general information and meta information of the project file. TwinCAT uses this information to create keys in the **Properties** tab. For example, if the **Company** text field contains the name "Company_A", the **Properties** tab contains the **Company** key with the value "Company_A".

Common Configuration: N/A Platform: N/A

Compile
Licenses
Statistic
SFC
Visualization
Visualization Profile
Static Analysis Light
Deployment
Compiler Warnings
UML
Advanced

Project information

Company:

Title:

Version: Released

Library Categories: ...

Default namespace:

Placeholder:

Author:

Description:

Library features

Global version structure:


POUs for property access:

Documentation format:

General

Minimize Id changes in TwinCAT files

Project information

For a library project, a company, title and version must be entered in order to be able to install the library.	
Company	Name of the company, which created this project (application or library). In addition to the library category, it is used for sorting in the library repository
Title	Project title
Version	Project version, e.g. "0.0.0.1"
Released	<input checked="" type="checkbox"/> : Protection against modification enabled. Consequence: When you now edit the project, a prompt will appear asking you if you really want to change the project. If you answer this query once with Yes, the prompt will no longer appear when the project is edited again.
Library Categories	Categories of the library project by which you can sort in the Library Repository dialog. If no category is specified, the library is assigned the category "Other". To assign it to another category, the category must be defined. Library categories are defined in one or several external description files in XML format. To assign the library, you can either call such a file or another library file that has already picked up information about the categories from a description file. Requirement: The project is a library project.
	The Library Categories dialog opens, in which you can add library categories.
Default namespace	The default setting for the namespace of a library is the library title. Alternatively, a different namespace can be defined explicitly, either generally for the library in the project information during library generation, or in the Properties dialog of the library reference for local use of the library in a project. The namespace of the library must be used as prefix of the identifier, in order to enable unambiguous access to a module that exists more than once in the project, or if the use of this prefix is enforced by the library property LanguageModelAttribute "qualified-access-only" ("Unambiguous access to library modules or variables"). If you do not define a standard namespace here, the name of the library file is automatically used as the namespace.
Placeholder	At this point, a default name for the placeholder can be specified, which represents or references this library. If a placeholder is not specified explicitly at this point, the default setting for the placeholder name of a library corresponds to the library title.
Author	Project author
Description	Brief description of the project (e.g. content, functionalities, general information such as "for internal use only", etc.)

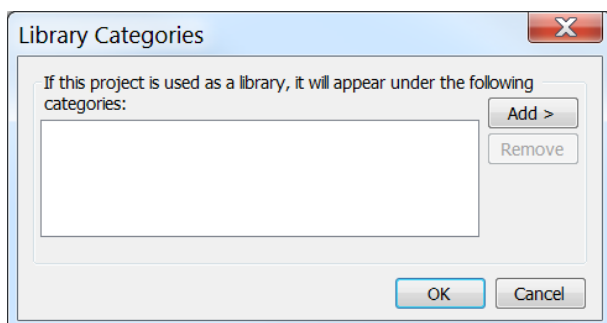
Library features

Creating a global version structure	Creates a global variable list in the PLC project, which contains the version information.
Automatically generate library information POU	Add button: POU objects of type "Function" are automatically created in the project tree, which can be used to access project properties in the application program. In this case, special functions are generated for the properties Company , Title and Version (F_GetCompany, F_GetTitle, F_GetVersion). If these functions have been added to the project by clicking Add , they can be removed from the project by clicking Remove .
Documentation format	Options: <ul style="list-style-type: none"> • Up to TC3.1 Build 4024: reStructuredText: • From TC3.1 Build 4026: TcDocGen During library creation, comments that correspond to a specific format are restructured and displayed in the Documentation tab of the Library Manager in this customized view. This opens up additional options for library documentation.
Allow implicit checks for compiled libraries	Available from TC3.1 Build 4026 <input checked="" type="checkbox"/> : TwinCAT executes implicit checks also for function blocks from protected libraries (*.compiled-libraries). Requirement: The compiler definition "checks_in_libs" is entered in the PLC project properties in the field Compiler definitions (category Compile). See also: Using function blocks for implicit checks [► 163]
Force Qualified_only for library access	Available from TC3.1 Build 4026 <input checked="" type="checkbox"/> : Objects from this library may only be used with the specification of the namespace of the library. See also: Attribute 'qualified_only' [► 822]
Allow referencing as library	Available from TC3.1 Build 4026 <input checked="" type="checkbox"/> : You can reference the PLC project in another PLC project as a library. See also: Use PLC project as referenced library

General

Minimize ID changes in TwinCAT files	<input checked="" type="checkbox"/> : The GUIDs of the PLC objects (e.g. POUs) are linked to those of the PLC project (using XOR). This avoids changes to the GUIDs of the PLC objects if they are used several times in different projects.
--------------------------------------	--

Library Categories dialog



List of categories	List of categories assigned to the library project. They can come from several sources. When you have entered all the desired categories, confirm the dialog with OK .
Add	The commands From Description File... and From Other Library... appear.
Remove	TwinCAT removes the selected category.
From description file...	The Select Description File dialog appears, in which you can select a description file with the extension *.libcat.xml. The file contains command categories. When you exit the dialog with Open , TwinCAT applies these categories.
From another library....	The Select Library dialog appears, in which you can select a library (*.library) whose command categories are to be adopted. When you exit the dialog with Open , TwinCAT applies these categories.
OK	TwinCAT provides the categories as project information and displays them in the Library Categories field.
Cancel	Closes the dialog. The process is aborted.

See also:

- PLC documentation: [Configuring a PLC project \[► 60\]](#)
- PLC documentation: [Using libraries \[► 266\]](#)

17.4.4.2 Category Compile

● Scope of PLC project properties



Note that the scope differs between different project properties!

Some properties affect only the PLC project whose properties you are currently configuring. Other properties, on the other hand, affect all PLC projects in the development environment. Such properties, which you can change in the project properties of a PLC project, but which also affect all other PLC projects, are titled **Solution options**.

The category **Compile** is used to configure the compiler options.

Settings

<p>Compiler definitions</p>	<p>Here you can enter compiler definitions/"defines" (see {define} statements) and conditions for compiling the application (conditional compilation).</p> <p>A description of the available conditional pragmas can be found in section Conditional pragmas [► 837]. The expression expr used in such pragmas can also be entered here. Several entries are possible in the form of a comma-separated list.</p>
<p>System compiler definitions</p>	<p>Available from TwinCAT 3.1 Build 4024</p> <p>The compiler definitions that have been set at System Manager level in the PLC project settings under Compiler definitions [► 924] are automatically adopted here.</p>
<p>Download application info</p>	<p>Available from TwinCAT 3.1 Build 4024</p> <p>Situation: You are loading a PLC project onto the controller that differs from the project already located there. In this case, a message window appears containing a Details button. Use this button to open the Application information window, which allows you to check the differences between the current PLC project and the PLC project on the controller. This involves comparing the number of function blocks, the data and the storage locations.</p> <p>The Application information window contains a brief description of the differences, for example:</p> <ul style="list-style-type: none"> • Declaration of MAIN changed • Variable fbMyNewInstance inserted in MAIN • Number of methods/actions of FB_Sample changed <p><input checked="" type="checkbox"/> (Default): If this setting is enabled, the information on the contents of the PLC project is loaded onto the PLC. This enables an extended check of the differences between the current PLC project and the PLC project on the controller. The difference in the extended check option is that the Application information window contains an additional Online comparison tab, which shows a tree comparison view. This will tell you which POUs have been changed, deleted or added. The additional tab appears when you execute the blue underlined command in the lower area of the Application information window ("Application not current. Generate code now to display the online comparison?").</p>

Generate tpy file	<p>Available from TwinCAT 3.1 Build 4024</p> <p>The tpy file contains, among other things, project, routing, compiler and target system information. It is the format used for describing a TwinCAT 2 PLC project. To ensure compatibility with existing applications, this file can be created for a TwinCAT 3 PLC project, if required.</p> <p><input type="checkbox"/> (Default): When the PLC project is created, no tpy file associated with the project is created.</p> <p><input checked="" type="checkbox"/> : When the PLC project is created, a tpy file associated with the project is created and stored in the project folder.</p> <p>Note that the value and configuration availability of this option depends on whether or not the TPY file is configured as a target file (see Settings tab ▶ 926).</p> <ul style="list-style-type: none"> • If the TPY file is enabled as a target file, the following happens: <ul style="list-style-type: none"> ◦ TwinCAT remembers the current status of the "Generate tpy file" option (= "original value", see below.). ◦ If this is not already the case, the option "Generate tpy file" is automatically activated next time the project is created. ◦ In addition, the "Generate tpy file" option is grayed out so that it cannot be disabled by the user as long as the TPY file is configured as a target file. • If the TPY file is subsequently disabled as a target file, the following happens: <ul style="list-style-type: none"> ◦ Next time the project is created, the "Generate tpy file" option is assigned its "original value" (see above.). ◦ In addition, the option is no longer grayed out, making it available again for configuration by the user.
Add comments about POU and DUT declarations to the TMC file	<p>Available from TwinCAT 3.1 Build 4026</p> <p><input checked="" type="checkbox"/> (Default): Comments on POU's and DUT's that are above their declaration are stored with the data type in the TMC file.</p> <p><input type="checkbox"/> : Comments above the declaration of POU's and DUT's are not stored in the TMC file. This will reduce the size of the TMC file if the comments contain detailed descriptions of the POU's and DUT's.</p>

Solution Options

<p>Compiler Version</p>	<p>Defines the compiler version that TwinCAT uses during compilation and during loading for compilation.</p> <p>Note that this setting does not replace the Remote Manager. For handling different engineering versions, if the PLC project is an application project, the Remote Manager should always be used. The compiler version should always be set to "latest" in this case.</p> <p>The compiler version setting is only relevant if the PLC project to be version-managed is a library project.. It is recommended that the library is saved with the oldest version with which it is ultimately to be used. To this end, the compiler version must be set to the corresponding fixed version (e.g. "3.1.4018.0").</p>
<p>Maximum number of warnings</p>	<p>Refers to the maximum number of warnings that TwinCAT issues in the Error List view.</p> <p>The selection of displayed compiler warnings is defined in the category Compiler warnings in the Project Settings dialog.</p>
<p>Replace constants</p>	<p><input checked="" type="checkbox"/> : TwinCAT loads the value directly for each constant of scalar type, i.e. not for STRING, ARRAY or structures. In online mode, TwinCAT identifies the constants in the declaration editor or monitoring window with a symbol preceding the value. In this case, access via an ADR operator, forcing or writing, for example, is not possible.</p> <p><input type="checkbox"/> (Default): Access to constants is possible. The computing time increases slightly.</p>

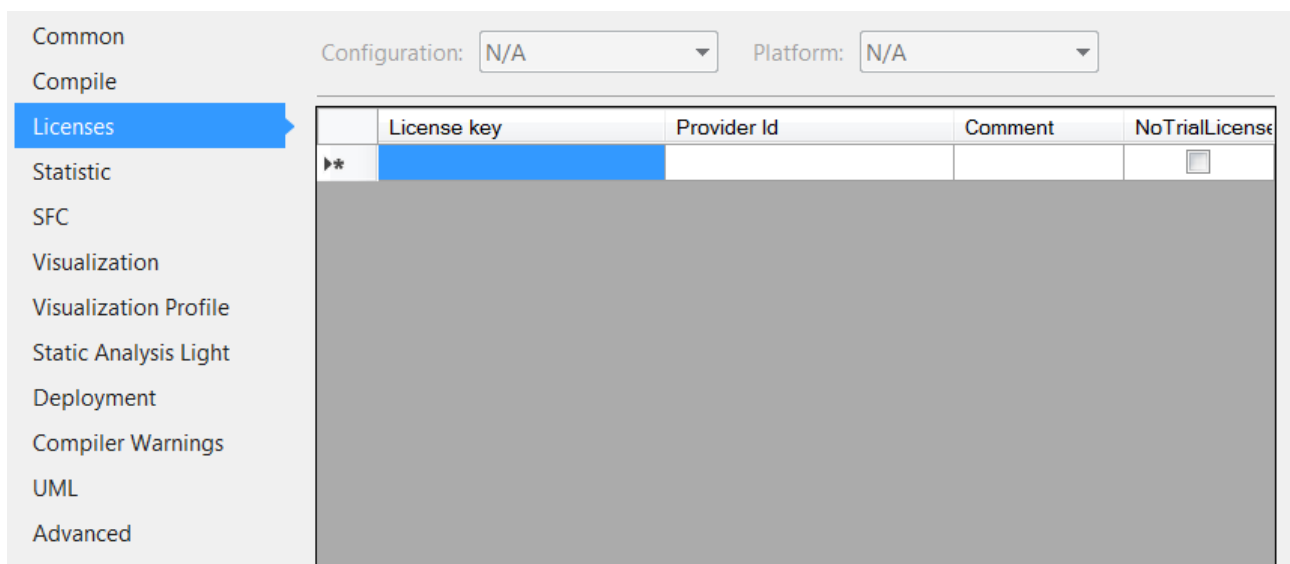
See also:

- [Category Compiler Warnings \[► 919\]](#)

17.4.4.3 Licenses category

In future, the **Licenses** category is intended to facilitate allocation of a TwinCAT 3 OEM license to a custom or proprietary library. This feature is not yet implemented.

This category is therefore not yet supported in the current TwinCAT version (reserved for future use).



The user must currently query an OEM license for his own library in the code of the library. See Query of an OEM license in the PLC application.

17.4.4.4 Category Statistic

The category **Statistic** provides statistical information on how many objects of the different types are present in the project.

Object type	Count
Action	2
Folder	4
Method	2
POU	3
Referenced Task	1

17.4.4.5 Category SFC

● Scope of PLC project properties



Note that the scope differs between different project properties!

Some properties affect only the PLC project whose properties you are currently configuring. Other properties, on the other hand, affect all PLC projects in the development environment. Such properties, which you can change in the project properties of a PLC project, but which also affect all other PLC projects, are titled **Solution options**.

The category **SFC** is used to configure the settings for SFC objects. Each new SFC object automatically has the configured settings in its properties.

Flags tab

Use	Variable	Declare	Description
<input type="checkbox"/>	SFCInit	<input checked="" type="checkbox"/>	All steps and actions are reset. The init step is activated. No actions will be executed.
<input type="checkbox"/>	SFCReset	<input checked="" type="checkbox"/>	All steps and actions are reset. The init step is activated and its actions will be executed.
<input type="checkbox"/>	SFCError	<input checked="" type="checkbox"/>	Gets 'TRUE', if a time check failed.
<input type="checkbox"/>	SFCEnableLimit	<input checked="" type="checkbox"/>	Enable time check on steps
<input type="checkbox"/>	SFCErrorStep	<input checked="" type="checkbox"/>	Contains the name of the step that caused SFCError to be 'TRUE'. SFCError is required.
<input type="checkbox"/>	SFCErrorPOU	<input checked="" type="checkbox"/>	Contains the name of the POU that caused SFCError to be 'TRUE'. SFCError is required.
<input type="checkbox"/>	SFCQuitError	<input checked="" type="checkbox"/>	Execution is stopped. SFCError is reset. SFCError is required.
<input type="checkbox"/>	SFCPause	<input checked="" type="checkbox"/>	Execution is stopped. SFCError is reset.
<input type="checkbox"/>	SFCTrans	<input checked="" type="checkbox"/>	Gets 'TRUE', if a transition switches through.
<input type="checkbox"/>	SFCCurrentStep	<input checked="" type="checkbox"/>	Contains the name of the active step.
<input type="checkbox"/>	SFCTip	<input checked="" type="checkbox"/>	Switches the next transition on a rising edge.
<input type="checkbox"/>	SFCTipMode	<input checked="" type="checkbox"/>	If 'TRUE', transitions can only be switched by means of SFCTip.
<input type="checkbox"/>	SFCErrorAnalyzation	<input checked="" type="checkbox"/>	Contains the possible variables that caused SFCError to be 'TRUE' in a string representation. SFCError is required
<input type="checkbox"/>	SFCErrorAnalyzationTable	<input checked="" type="checkbox"/>	Contains the possible variables that caused SFCError to be 'TRUE' in a table. SFCError is required

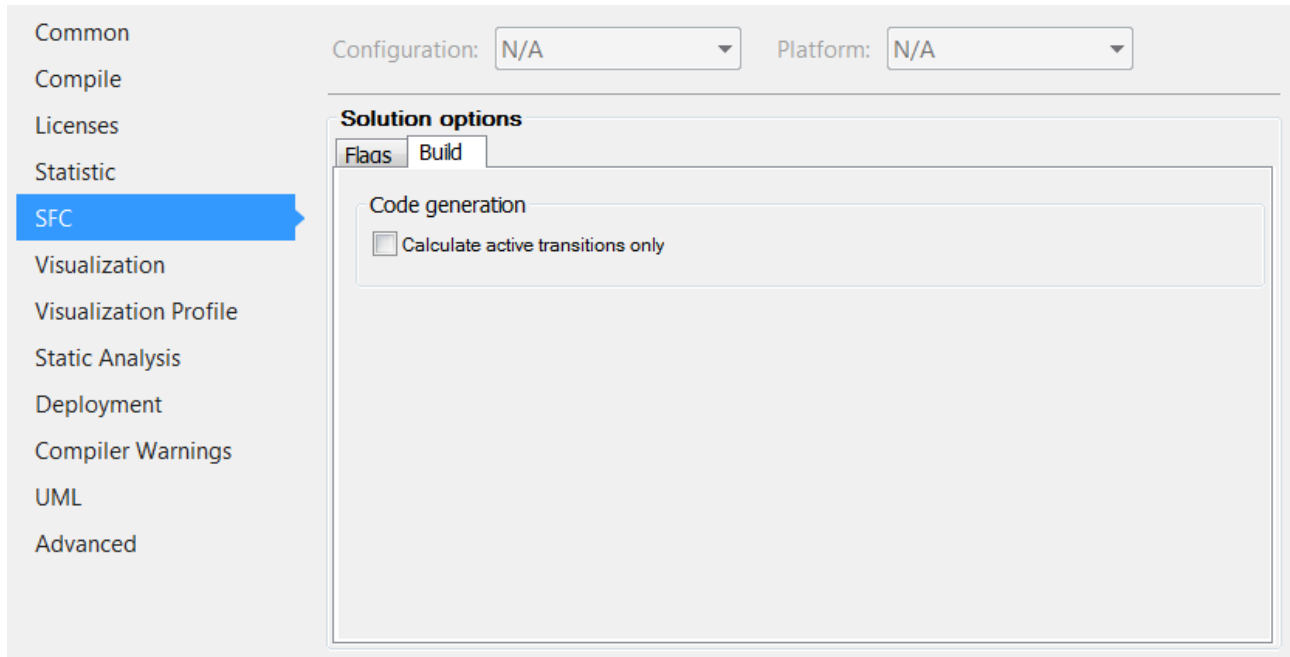
Implicitly generated variables (flags) for controlling and monitoring the processing in an SFC diagram.

Use	: The corresponding variable is used.
Declare	: The corresponding variable is created automatically. Otherwise, if the usage is intended (Use is set), the user has to declare the variable.




An automatically declared flag variable appears in the declaration part of the SFC editor, but only in online mode.

Build tab



Code generation

Calculate active transitions only	 : TwinCAT generates code only for transitions that are currently active.
-----------------------------------	--

See also:

- [SFC Flags \[▶ 639\]](#)

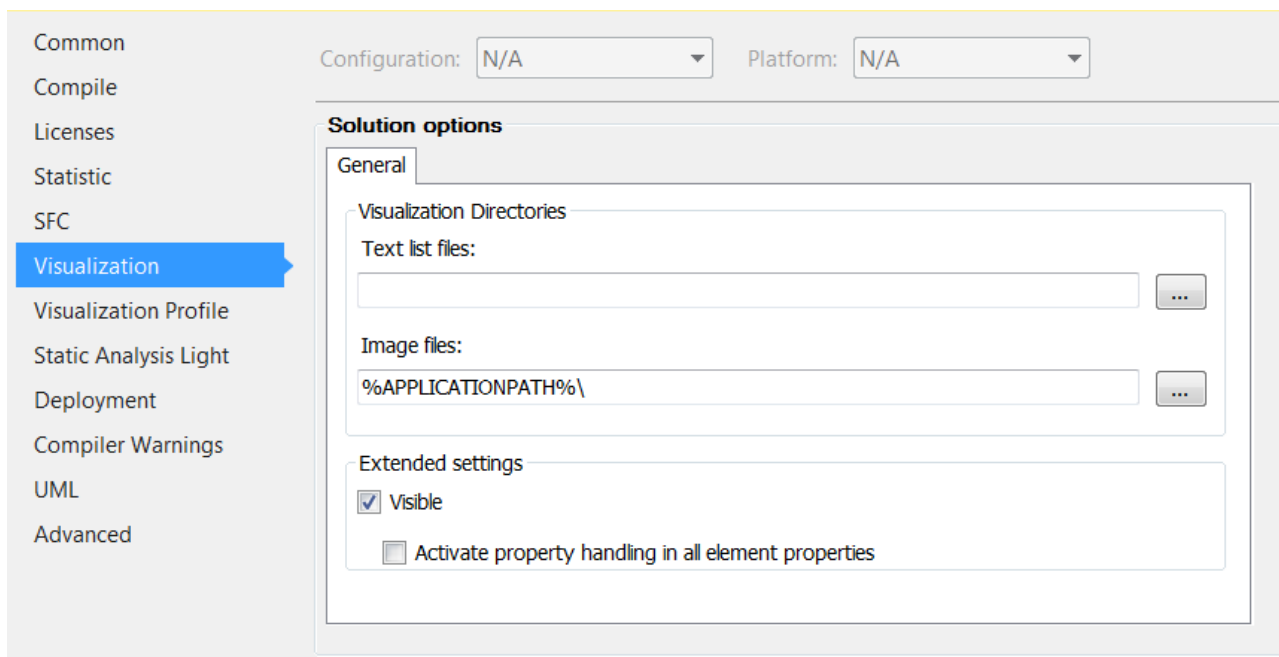
17.4.4.6 Category Visualization



Scope of PLC project properties

Note that the scope differs between different project properties! Some properties affect only the PLC project whose properties you are currently configuring. Other properties, on the other hand, affect all PLC projects in the development environment. Such properties, which you can change in the project properties of a PLC project, but which also affect all other PLC projects, are titled **Solution options**.

The category **Visualization** is used to configure the project-wide settings for objects of type Visualization.



General tab

Visualization Directories

Text list files	<p>Directory containing text lists available in the project for configuring texts for different languages. TwinCAT uses this directory when exporting or importing text lists, for example.</p> <p>Click to bring up the Find Folder dialog, which allows you to select a directory in the file system.</p>
Image files	<p>Directory containing image files available in the project. Multiple folders are separated by semicolons. TwinCAT uses this directory when exporting or importing image files, for example.</p> <p>Click to bring up the Find Folder dialog, which allows you to select a directory in the file system.</p>

Extended settings

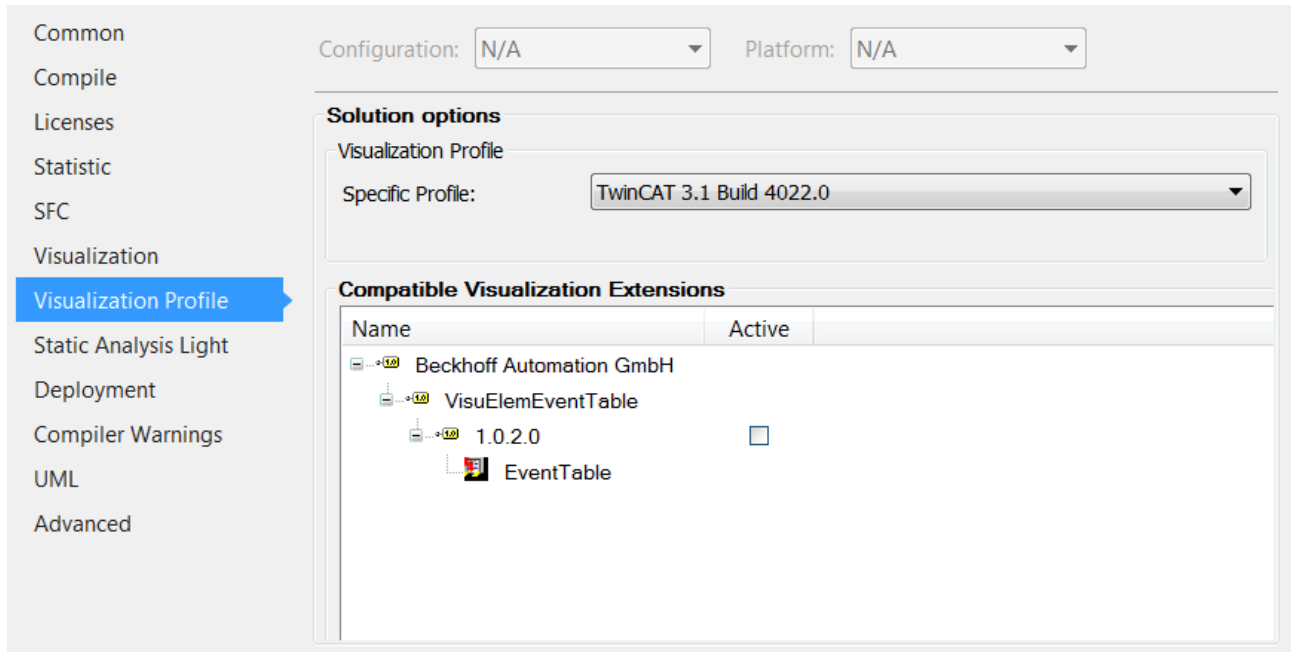
Enables property handling in all element properties	<p> : You can also configure a visualization element in those of its properties, in which you select an IEC variable, with a property. TwinCAT then generates additional code for properties handling when compiling a visualization.</p> <p>Requirement: Your IEC code contains at least one object of type Interface property, i.e. a property .</p> <pre> ▲ MAIN (PRG) ▲ Property_A Get Set </pre> <p>Requirement: Visible is enabled.</p>
---	---

17.4.4.7 Category Visualization Profile

● Scope of PLC project properties

i Note that the scope differs between different project properties! Some properties affect only the PLC project whose properties you are currently configuring. Other properties, on the other hand, affect all PLC projects in the development environment. Such properties, which you can change in the project properties of a PLC project, but which also affect all other PLC projects, are titled **Solution options**.

The **Visualization Profile** category allows you to set the visualization profile.



Visualization profile

Specific Profile	Profile, which TwinCAT uses in the project and which determines the visualization elements that are available in the project. The selection list contains all previously installed profiles.
------------------	---

17.4.4.8 Category Static analysis

● Scope of PLC project properties

i Note that the scope differs between different project properties! Some properties affect only the PLC project whose properties you are currently configuring. Other properties, on the other hand, affect all PLC projects in the development environment. Such properties, which you can change in the project properties of a PLC project, but which also affect all other PLC projects, are titled **Solution options**.

The category **Static analysis** defines the checks that are taken into account in the static code analysis.

Static Analysis Light:

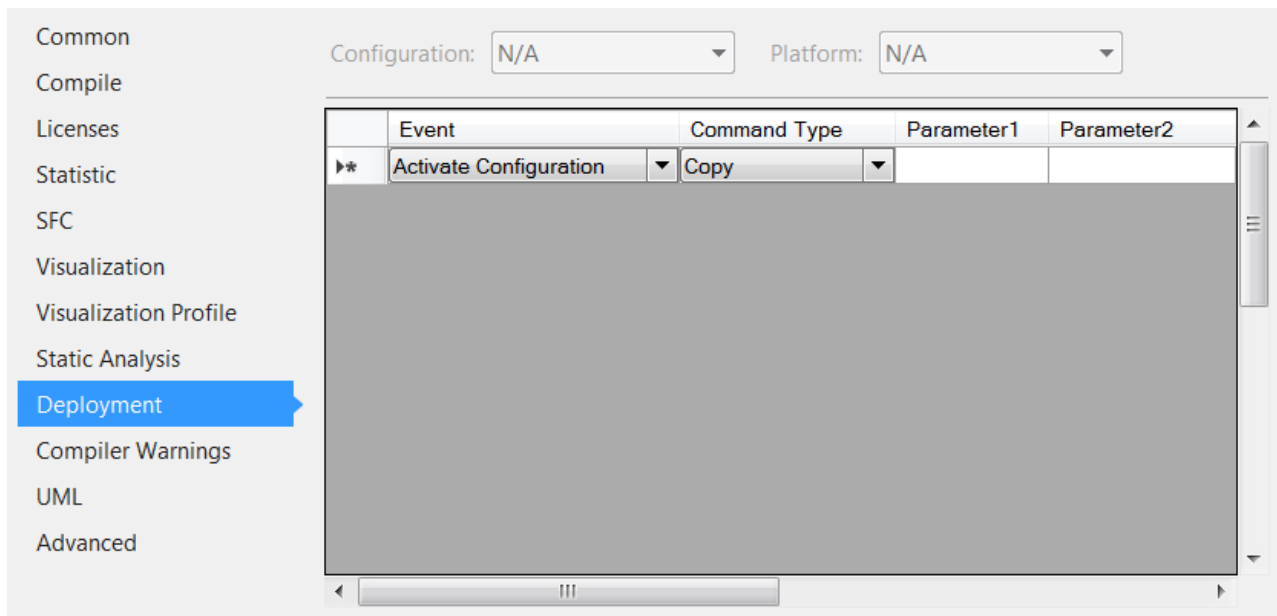
- If you have not activated the additional TE1200 Engineering license, you can use the license-free version of Static Analysis (Static Analysis Light), which contains some coding rules. The free Light version enables you to familiarize yourself with the basic handling of the product, for example, based on a reduced set of functions.
- For more information about Static Analysis Light see:
PLC documentation: Programming a PLC Project > Checking syntax and analyze code > [Code analysis \(Static Analysis\)](#) [▶ 148]

Static Analysis Full:

- If you have enabled the additional TE1200 Engineering license, the full range of Static Analysis functions is available (saving and loading settings, more than 100 coding rules, naming conventions, metrics, forbidden symbols).
- For more information about Static Analysis Full see: TE1200 Static Analysis.

17.4.4.9 Category Deployment

The category **Deployment** is used to set up commands that are to be executed during the installation and startup of an application.



The following events are available, after which the commands listed in the list can be called:

Activate Configuration	The required command is called up after the configuration has been enabled.
Plc Download	The required command is called up after the PLC application has been downloaded to the target system.
Plc Online Change	The required command is called after a successful online change.
Plc After Compile	The required command is called after a compilation of the PLC application.

The following commands can be executed:

Copy	Copies files from parameter 1 (source path) to a location specified in parameter 2 (target path).
Execute	Executes the application or script listed under parameter 1.

Source and target paths can contain virtual environment variables, which TwinCAT resolves accordingly.

The following environment variables are supported:

Virtual environment variable	Registry value	Default value
%TC_INSTALLPATH%	InstallDir	C:\TwinCAT\3.x \
%TC_TARGETPATH%	TargetDir	C:\TwinCAT\3.x \Target\
%TC_BOOTPRJPATH%	BootDir	C:\TwinCAT\3.x \Boot\
%TC_RESOURCEPATH%	ResourceDir	C:\TwinCAT\3.x \Target\Resource\
%SOLUTIONPATH%	-	Location of the solution file

Registry values are stored in the registry under the following key: `\HKLM\Software\Beckhoff\TwinCAT3`.

Example:

In the following example, the file *SampleFile.xml* file is copied from the Config project subfolder of the solution to the folder *C:\plc\config* on the target system.

Event	Command Type	Parameter1	Parameter2
Activate Configuration	Copy	%SOLUTIONPATH%\Config\SampleFile.xml	C:\plc\Config\SampleFile.xml

17.4.4.10 Category Compiler Warnings

The **Compiler Warnings** category is used to select the compiler warnings that TwinCAT displays in the message window during a compilation run.



You can specify the maximum number of listed warnings in the **Compile** category.

See also:

- [Command Build PLC project \[► 929\]](#)
- [Category Compile \[► 910\]](#)

17.4.4.11 Category UML



Scope of PLC project properties

Note that the scope differs between different project properties! Some properties affect only the PLC project whose properties you are currently configuring. Other properties, on the other hand, affect all PLC projects in the development environment. Such properties, which you can change in the project properties of a PLC project, but which also affect all other PLC projects, are titled **Solution options**.

In the category **UML** you can change the UML compiler version. This setting is only relevant when using the UML Statechart.

For more information on the configuration options, please refer to section "UML Compiler Version" of the TF1910 TC3 UML documentation.

The screenshot shows the 'UML' category selected in the left sidebar. The main area displays 'Solution options' with the following table:

Solution options	
UML compiler version in project	4.0.2.1
Recommended, newest version	4.0.2.1
Action	Do not update.

17.4.4.12 Category Advanced

● Scope of PLC project properties



Note that the scope differs between different project properties!

Some properties affect only the PLC project whose properties you are currently configuring.

Other properties, on the other hand, affect all PLC projects in the development environment. Such properties, which you can change in the project properties of a PLC project, but which also affect all other PLC projects, are titled **Solution options**.

The category **Advanced** is used to configure advanced properties.

Write options

● Engineering incompatibility of file version 1.2.0.0 (or higher) with TwinCAT 3.1 < build 4024



Please note that objects saved with file version 1.2.0.0 (or higher) cannot be loaded with Engineering versions lower than TwinCAT 3.1.4024!

Since an object is automatically saved with file version 1.2.0.0 when using the optional Base64 format, objects with Base64 format cannot be loaded with Engineering versions lower than TwinCAT 3.1.4024.

If a PLC project contains objects with the file version 1.1.0.1 and objects with the file version 1.2.0.0, the 1.1.0.1 objects are loaded with an Engineering version lower than TwinCAT 3.1.4024. Objects with file version 1.2.0.0 are not loaded.

The file version of a file saved with file version 1.2.0.0 can be reset to 1.1.0.1 using XAE version TwinCAT 3.1.4024 or higher.

Write object content as	<p>Available from TwinCAT 3.1 Build 4024</p> <p>Background information:</p> <p>From Build 4024, Base64 introduces a new storage format that is optionally available for the following PLC objects:</p> <ul style="list-style-type: none"> • POU's where the POU body is programmed in a graphical implementation language <ul style="list-style-type: none"> ◦ Sequential Function Chart (SFC) ◦ FBD/LD/IL (Function Block Diagram/Ladder Diagram/Instruction List) ◦ CFC (Continuous Function Chart and page-oriented CFC) ◦ UML class diagram and state diagram • POU's with a subelement (e.g. action, method) that is programmed in a graphical implementation language (for graphical languages see first key point) • Visualizations • Visualization Manager • Text lists • Recipe manager • Image pool <p>Up to now, these objects were saved as XML by default.</p> <p>From Build 4024 you can configure whether these object types should be saved as XML or Base64.</p> <p>Advantages of Base64 over XML:</p> <p>Base64 results in compressed storage, compared to XML. As a result, improved performance can be achieved with file access to these objects, which can be used, for example, when loading projects or moving/copying objects.</p> <p>Setting option for the standard storage format:</p> <p>For a PLC project, the setting "Write object content as" in the PLC project properties can be used to define the standard storage format for the object types mentioned above.</p> <p>The selected standard storage format is only used with newly added objects (exception: not with newly added POU sub-objects. Example: A POU is saved as an XML and the standard storage format is configured as Base64. If a graphic sub-object is then added to the POU, the storage format of the POU and thus of the sub-object as XML remains unchanged).</p> <p>The storage format of an existing object with a non-standard storage format is not automatically changed when the object is changed and saved. The storage format of an existing object can be changed individually via the Properties window (see below). Alternatively, when changing the standard storage format, there is the option of adopting the newly selected storage format for all existing objects. If you change the storage format at this point, a corresponding query window appears.</p> <p>The following options are available for the setting "Write object content as":</p> <ul style="list-style-type: none"> • XML (default): The PLC objects mentioned above are saved in XML format by default. <ul style="list-style-type: none"> ◦ Objects with this storage format are stored in file version 1.1.0.1. • Base64: The PLC objects mentioned above are saved in Base64 format by default. <ul style="list-style-type: none"> ◦ Objects with this storage format are stored in file version 1.2.0.0. Please note that objects with the file version 1.2.0.0 (or higher) cannot be loaded with Engineering versions < TwinCAT 3.1.4024! <p>Individual setting option of the storage format:</p> <p>The storage format of an object can be configured individually for the object types mentioned above in the Properties window of the object. For more information see the description of the properties [► 901] (Format property).</p>
-------------------------	---

Write product version in files	<p>Available from TwinCAT 3.1 Build 4024</p> <p>The product version indicates which plug-in version was used to save a PLC file (e.g. a function block). The setting of this checkbox is valid for the whole project and is the default setting for all modified or newly added PLC objects located in this PLC project.</p> <p><input checked="" type="checkbox"/> (Default): The product or plugin version is written into the file (the version is not visible in XAE; it shows up if the file is analyzed at file level).</p> <ul style="list-style-type: none"> • If you change the setting from disabled to enabled, a query window appears in which you can select whether to add the product version to all existing files. • Use case for the enabled option: This setting can be used to include the file version in the file for debugging or tracking purposes, for example. • Please note the following: If the file is saved with a different product version, this leads to a change of this file, which shows up as a file difference when using source code management systems. <p><input type="checkbox"/> : The product or plugin version is not written to the file.</p> <ul style="list-style-type: none"> • If you change the setting from enabled to disabled, a query window appears in which you can select whether to remove the product version from all existing files. • Use case for the disabled option: This setting can be used if the product version is not of interest. This minimizes changes to files with regard to source code management systems.
Write object content with profile	<p>The profile defines the format in which objects are saved. With Build 4024, for example, new functionalities were added for the PLC HMI. For this reason, visualization files saved with Build 4024 cannot be directly opened with older builds. If you set a 4022 profile here, then the visualization files will be saved in the appropriate format and can be opened with Build 4022.</p> <p>Requirement: So that, for example, the 4022 profile is available in the drop-down menu, either a 4022 Remote Manager installation must be carried out or the current 4024 XAE installation must have been installed via a previously existing 4022 XAE installation.</p>
Write Bookmarks to File	<p>Available from TwinCAT 3.1 Build 4026</p> <p><input checked="" type="checkbox"/> : The stored bookmarks are also written from the Visual Studio project user options file (.suo) to a separate file. This file ends with .bookmarks and is located in the project directory. It is then also part of the known archive options.</p> <p><input type="checkbox"/> (default setting): The bookmarks are only saved in the .suo file from Visual Studio. A .bookmarks file that has already been created is deleted from the project directory.</p> <p>The global default setting for new PLC projects as to whether bookmarks should be stored in a separate file can be found at Dialog Options - Write Options [► 994]. The value is transferred once to this local project setting when a new PLC project is created.</p>

Multiuser options

Use Multiuser	<p>Available from TwinCAT 3.1 Build 4024</p> <p><input type="checkbox"/> (Default): The multiuser functionality of the PLC project is not enabled.</p> <p><input checked="" type="checkbox"/> : The multiuser functionality of the PLC project is enabled.</p> <p>Please also refer to the further information in the Multiuser documentation.</p>
---------------	--

Solution options

<p>Secure Online Mode</p>	<p><input type="checkbox"/> : (Default): For security reasons, the user is always prompted to confirm the execution of the following commands when they are called.</p> <ul style="list-style-type: none"> • Activate the configuration • Restart TwinCAT System in Config/Run Mode • Reset cold • Reset origin <p><input checked="" type="checkbox"/> : In addition to the above commands, for which a confirmation prompt appears by default, the following commands will also prompt you to confirm.</p> <ul style="list-style-type: none"> • Start • Stop • Single Cycle
<p>Autoupdate Visu Profile</p>	<p>This option allows you to configure the automatic update behavior of the visualization profile.</p> <p>When you open a PLC project that uses an outdated visualization profile, a warning appears in the message window ("New Version found for Visualization profile").</p> <p><input checked="" type="checkbox"/> : In such a case, the visualization profile version is automatically set to the latest version if the option Autoupdate Visu Profile is enabled. With such an automatic update of the visualization profile version, a corresponding warning is displayed in the message window (e.g. "Visualization profile set from' TwinCAT 3.1 Build 4020.10' to' TwinCAT 3.1 Build 4022.0").</p> <p><input type="checkbox"/> (Default): If the Autoupdate Visu Profile option is disabled, the visualization profile version is not changed automatically. By double-clicking on the warning "New Version found for Visualization profile", you can open the ProfileUpdate dialog in which you can manually change the visualization profile version.</p>
<p>Autoupdate Uml Profile</p>	<p>This option allows you to configure the automatic update behavior of the UML compiler version.</p> <p>If you open a PLC project, in which an outdated UML compiler version is used, a corresponding warning appears in the message window ("new version for UML found").</p> <p><input checked="" type="checkbox"/> : In such a case, the UML compiler version is automatically set to the latest version if the option Autoupdate Uml Profile is enabled. In the case of such an automatic update of the UML compiler version, a corresponding warning will be displayed in the message window (e.g. "UML set from '4.0.2.0' to '4.0.2.1").</p> <p><input type="checkbox"/> (Default): If the option Autoupdate Uml Profile is disabled, the UML compiler version is not changed automatically. Double-click on the warning "new version for UML found" to open the ProfileUpdate dialog, in which you can change the UML compiler version manually.</p> <p>For more information, see UML Compiler Version.</p>

Write Line IDs	<p>Available from TwinCAT 3.1 Build 4026</p> <p><input checked="" type="checkbox"/> : Line IDs are generated and stored for the POU's of the project (default behavior up to TwinCAT 3.1 Build 4024). The line IDs can be used to assign lines of code to machine code instructions, which is required for breakpoint handling, among other things.</p> <p><input type="checkbox"/> (Default): No separate line IDs are generated. For the assignment of the machine code instructions and the breakpoint handling in this case the line number is used. Therefore, an online change is required for changes such as spaces or comments.</p> <p>The global default setting for new PLC projects regarding the Write Line IDs can be found at Dialog Options - Write Options [▶ 994]. The value is transferred once to this local project setting when a new PLC project is created.</p>
----------------	---

Compatibility

Convert PLC Project to previous TwinCAT version	<p>Available from TwinCAT 3.1 Build 4026</p> <p>In the dialog that opens you can select a TwinCAT version to which the PLC project should be converted (TwinCAT 3.1 Build 4022 or 4024). After confirming the conversion with "Convert", the project is closed and saved compatible with the selected version.</p> <p>Note that the project data will be changed during the conversion and settings and properties of later versions will be lost. Therefore, the conversion is not suitable for multiple switching between different versions.</p>
---	---

17.4.5 PLC project settings

Function: This command opens an editor in which project settings can be defined.

Call: Double-click on the PLC project in the **Solution Explorer**

See also:

- PLC documentation: [Configuring a PLC project \[▶ 60\]](#)

17.4.5.1 Project tab

The screenshot shows the 'Project Settings' dialog box with the following fields and options:

- Project Name:** PlcSampleProject
- Id:** 1
- Project Path:** PlcSampleProject
- Project Type:** Plc Project
- Port:** 851
- Project Guid:** {64AB6063-1DA8-405D-A514-E441BF3F11D5}
- Encryption:** No boot project encryption (default)
- Autostart Boot Project
- Symbolic Mapping
- Force Multi Instance
- Comment:** (Empty text area)

Project Name	Name of the PLC project
Id	ID of the PLC project
Project Path	Path to the location where the PLC project is stored
Project Type	Project type
Port	AMS port number of the runtime system
Project Guid	GUID of the PLC project
Encryption	Encryption of the boot project <ul style="list-style-type: none"> • No boot project encryption (default) • Encrypt boot project
Autostart Boot Project	<input checked="" type="checkbox"/> After the TwinCAT runtime environment has been started, the PLC boot project is automatically loaded and started. The setting is transferred directly to the currently selected target system. The setting is not saved in the TwinCAT project. This option corresponds to the Autostart Boot Project command in the context menu of the PLC project node in the Solution Explorer.
Symbolic Mapping	<input checked="" type="checkbox"/> Symbolic mapping is enabled.
Force Multi Instance	<input checked="" type="checkbox"/> Option for logging in multiple instances of the PLC project enabled.
Comment	Comment box
Compiler Defines (available from TC3.1 build 4024)	
Manual	Here you can define your own compiler definitions at System Manager level, which are passed on to the PLC project. The definitions are entered in the PLC project properties under the category compile as <u>system compiler definitions</u> [▶ 910].
Implicit	<input checked="" type="checkbox"/> The names of the selected project variants as well as all groups to which the project variant belongs are automatically set as a compiler definition and passed on to the PLC project. The definitions are entered in the PLC project properties under the category compile as <u>system compiler definitions</u> [▶ 910]. Note: In order to activate this checkbox, the <u>Defines</u> for the variant management must be enabled.

See also:

- Variant management: Concept: [Integration in the PLC project](#)
- Command properties (PLC project): [Category Compile: System compiler definitions](#) [[▶ 910](#)]

17.4.5.2 Settings tab

The screenshot shows the 'Settings' tab with the following options:

- Target Archive:**
 - Login Information
 - Project Sources
 - Compiled Libraries
 - Source Libraries
- File/E-Mail Archive:**
 - Login Information
 - Project Sources
 - Compiled Libraries
 - Source Libraries
 - Core Dump
- Target Files:**
 - Boot Files
 - TMC File
 - TPY File
- Target Behavior:**
 - Clear Invalid Persistent Data

Target Archive

In the **Target Archive** group box you can specify which information is transferred to the target system together with other data when you create a boot project.

Login Information	COMPILEINFO file containing the compiler information of the PLC project.
Project Sources	Source code files of the PLC project in readable source code form.
Compiled Libraries	Libraries that are used in compiled form in the PLC project.
Source Libraries	Libraries that are used in legible source code form in the PLC project.

File/E-Mail Archive

In the **File/E-Mail Archive** group box, you can specify what information is stored when archiving a PLC project [[▶ 863](#)], a TwinCAT project [[▶ 1064](#)] or a Solution [[▶ 861](#)]. If you activate the corresponding checkbox, the files described in the following table are stored in the project archive.

Login Information	COMPILEINFO file containing the compiler information of the PLC project.
Project Sources	Source code files of the PLC project in readable source code form.
Compiled Libraries	Libraries that are used in compiled form in the PLC project.
Source Libraries	Libraries that are used in legible source code form in the PLC project.
Core dump	Core dump file, which is located in the PLC project directory, and the compile info files, which are located in the "_CompileInfo" folder in the project directory. Note: The compile info files are also saved in the archive if the " <u>Core Dump</u> [▶ 951]" setting is activated, as these files are needed in order to be able to use the core dump.

● Transfer of source code

i If you have configured the target or file/email archive settings to include the project sources and/or source libraries in one of these archives, please note that the project sources and/or the source libraries (*.library) used in the project are contained in the ZIP archive in readable source code form when passing on/delivering the target system or when passing on the file/email archive.

Keep this in mind when configuring the settings described above and when storing and referencing libraries (*.library vs *.compiled-library).

For more information on library management, see section [Using libraries](#) [▶ 266].

Information about source code encryption can be found in the documentation on [Security Management](#).

Target Files

In the **Target Files** group box you can set which information is transferred to the \Boot\Plc folder when you create a boot project on the target system.

TMC File	TMC file (TwinCAT Module Class) of a PLC project.
TPY File	TPY file (contains, among other information, project information, routing information, compiler information, target system information).

Target Behavior

Clear Invalid Persistent Data	The backup of the stored persistent data is ignored. This ensures that any invalid data is not accepted but instead discarded. See: Backup of persistent data [▶ 927]
-------------------------------	--

Backup of persistent data

Persistent data is regularly stored in a .bootdata file in the TwinCAT\Boot folder during a TwinCAT system stop/shutdown. At the next system startup (TwinCAT Run mode) this file is read, and the persistent variables in the runtime system are initialized with the values from the file. The system renames the .bootdata file to .bootdata-old.

This backup file (.bootdata-old) of the persistent data is read at system startup if the file (.bootdata) containing the persistent data does not exist. This is an exception, but it can occur, for example, if an IPC without UPS experiences a power failure and TwinCAT could not shut down properly.

- If it is foreseeable that the contents of the backup file will not be usable at a new system start, you can enable the option **Clear Invalid Persistent Data** to ignore the backup file. This can be the case, for example, if batch information or tool data has been stored in a production facility and has to be up-to-date.
- If the structure of the persistent data (its data types or symbol paths in the program code) is changed due to online changes, it makes no sense to subsequently load an obsolete persistent data file. In this case, you should enable the **Clear Invalid Persistent Data** option in advance.

In both cases, you should also ensure that a current persistent data file is available. Function blocks such as FB_WritePersistentData (PLC Lib Tc2_Uilities) and UPS protection against sudden power failures are available for this purpose.

When using persistent data, the corresponding flags (BootDataLoaded and OldBootData) from the global structure PlcAppSystemInfo should always be evaluated (see documentation on System > Global Data Types).


If neither the regular file nor the backup file can be loaded or if they don't exist, the variables marked as PERSISTENT are reinitialized in the same way as other "normal" variables, either with their explicitly specified initial values or with the standard initializations.

See also:

- PLC documentation: [Remanent Variables - PERSISTENT, RETAIN](#) [▶ 691]

17.5 Build

17.5.1 Command Build Solution

Symbol: 

Function: This command starts the compilation process or the code generation for all projects contained in the solution.

Call: **Build** menu or context menu of the solution

Requirement: The solution is selected.

All projects contained in a solution are compiled one after the other. This also concerns the projects (PLC, C++, etc.) integrated below a TwinCAT project. The steps performed for a PLC project are described in section [Command Build PLC project \[► 929\]](#).

17.5.2 Command Rebuild Solution

Function: The command starts the compilation process for all projects contained in a solution, even if it was previously compiled without errors.

Call: **Build** command or context menu of the solution

Requirement: The solution is selected.

When rebuilding the solution, it will first be cleaned (see also: [Command Clean Solution \[► 928\]](#)) and subsequently built (see also: [Command Build Solution \[► 928\]](#)).

See also:

- [Command Rebuild a PLC project \[► 930\]](#)

17.5.3 Command Clean Solution

Function: This command starts the cleaning of all projects contained in the solution.

Call: **Build** menu or context menu of the solution

Requirement: The solution is selected.

All projects contained in the solution are cleaned in succession. This also concerns the projects (PLC, C++, etc.) integrated below a TwinCAT project. The steps executed for a PLC project are described in the section [Command Clean PLC project \[► 930\]](#).

17.5.4 Command Check all objects

Function: The command initiates a compilation run, i.e. a syntax check for all objects located in the project tree of the PLC project. This is primarily useful when creating libraries or when processing library projects.


Call: Context menu of the PLC project object (<PLC project name> Project) in the **Solution Explorer**

As opposed to the [Command Build PLC project \[► 929\]](#), in which only the objects used are checked, when this command is executed the syntax of all objects in the PLC project is checked.



The command does not lead to code generation. No file with information on the compilation run is created in the project directory.

17.5.5 Command Build TwinCAT project

Symbol: 

Function: This command starts the compilation process or the code generation for the currently active TwinCAT project.

Call: **Build** menu if a TwinCAT project is currently selected, or context menu of the TwinCAT project

Requirement: The TwinCAT project is selected.

All of the projects (PLC, C++, etc.) contained in the TwinCAT project are compiled one after the other. The steps performed for a PLC project are described in section [Command Build PLC project \[▶ 929\]](#).

See also:

- [Command Rebuild a TwinCAT project \[▶ 929\]](#)

17.5.6 Command Rebuild a TwinCAT project

Function: The command starts the compilation process or the code generation for the currently active TwinCAT project, even if it was last compiled without error.

Call: **Build** menu if a TwinCAT project is currently selected, or context menu of the TwinCAT project

Requirement: The TwinCAT project is selected.

When rebuilding the project, the TwinCAT project will first be cleaned (see also: [Command Clean TwinCAT project \[▶ 929\]](#)) and subsequently built (see also: [Command Build TwinCAT project \[▶ 929\]](#)).

17.5.7 Command Clean TwinCAT project

Function: This command deletes the local compilation information for the currently active PLC project and updates the language model of all objects.

Call: **Build** menu if a TwinCAT project is currently selected, or context menu of the TwinCAT project


Requirement: The TwinCAT project is selected.

All of the projects (PLC, C++, etc.) contained in the TwinCAT project are cleaned one after the other. The steps executed for a PLC project are described in the section [Command Clean PLC project \[▶ 930\]](#).

See also:

- [Command Rebuild a TwinCAT project \[▶ 929\]](#)

17.5.8 Command Build PLC project

Symbol: 

Function: This command starts the compilation process or the code generation for the currently active PLC project.

Call: **Build** menu if a PLC project is currently selected, or context menu of the PLC project object (<PLC project name> Project) in the **Solution Explorer**

Requirement: The PLC project is selected.

During the compilation, TwinCAT carries out a syntactic test of all objects used in the PLC project. The compilation procedure is always carried out automatically when you wish to log the project in with a changed program. After the check has been completed, TwinCAT displays any error messages or warnings in the [Error List \[► 894\]](#) view.

Apart from that, the compilation information of the PLC project is created when building the project and saved in a local file (*.compileinfo) in the Solution.

If the program was not changed since the last error-free compilation process, it is not recompiled. If the syntactic test is to be performed anyway, use the [Command Rebuild a PLC project \[► 930\]](#).

17.5.9 Command Rebuild a PLC project

Function: The command starts the compilation process or the code generation for the currently active PLC project, even if it was last compiled without error.

Call: **Build** menu if a PLC project is currently selected, or context menu of the PLC project object (<PLC project name> Project) in the **Solution Explorer**

Requirement: The PLC project is selected.

When rebuilding the project, the project will first be cleaned (see also: [Command Clean PLC project \[► 930\]](#)) and subsequently built (see also: [Command Build PLC project \[► 929\]](#)).

17.5.10 Command Clean PLC project

Function: The command updates the language model for all of the objects in the currently active PLC project.

Call: **Build** menu if a PLC project is currently selected, or context menu of the PLC project object (<PLC project name> Project) in the **Solution Explorer**

Prerequisite: The PLC project is selected.

If the PLC project is cleaned up, only the language model for all objects in the PLC project is updated. The compilation information on the target system is retained.

See also:

- [Command Rebuild a PLC project \[► 930\]](#)

17.6 Debug

17.6.1 Command New Breakpoint

Symbol: 

Function: The command opens the **Breakpoint Properties** dialog.

Call: **Debug** menu, button  **New** in the **Breakpoint** view (**PLC > Window > Breakpoints**).

Requirement: The PLC project is in online mode.




The command **Toggle Breakpoint** can be used to set a new breakpoint directly at the current cursor position in online mode.

See also:

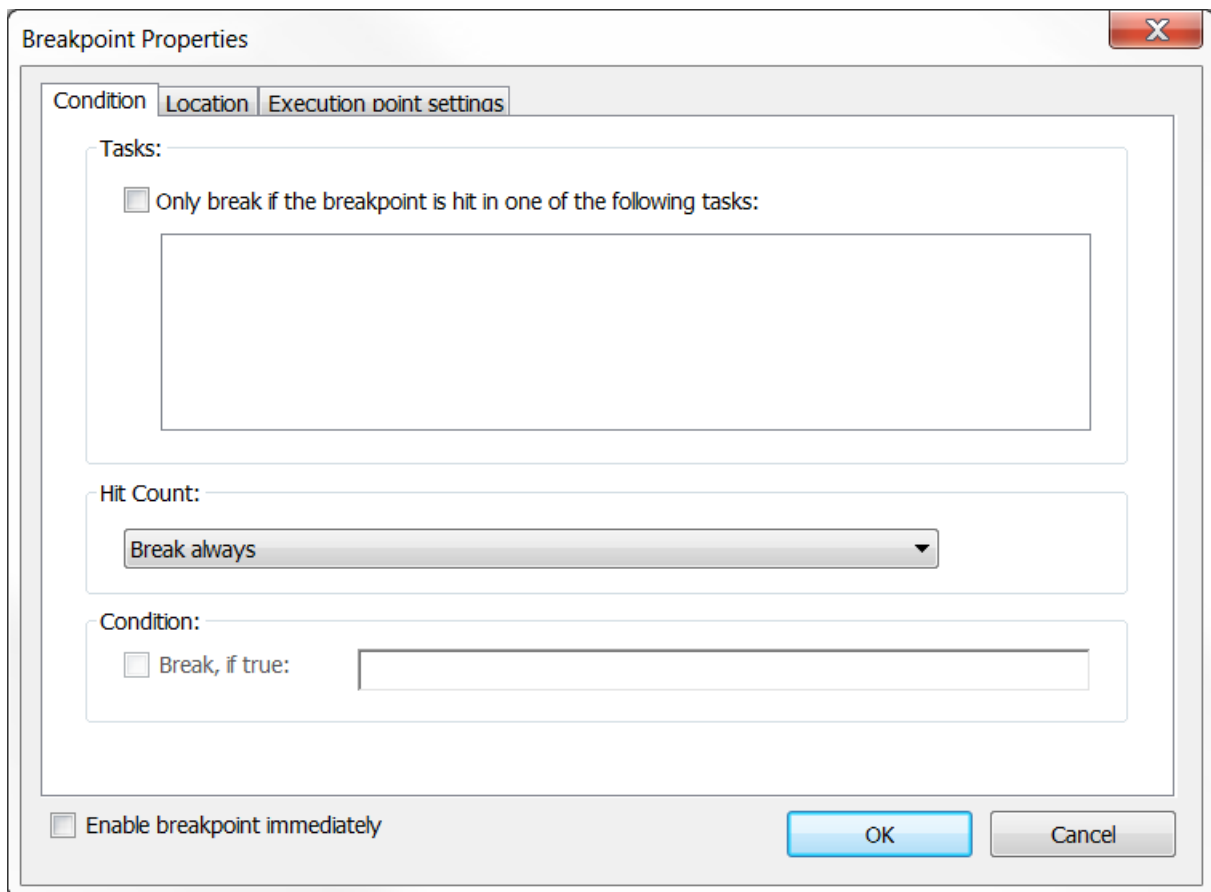
- [Command Toggle Breakpoint \[▶ 934\]](#)
- PLC menu: [Command Breakpoints \[▶ 943\]](#)
- PLC documentation: [Use of breakpoints \[▶ 211\]](#)

Breakpoint Properties dialog

<p>Enable breakpoint immediately</p>	<p><input checked="" type="checkbox"/> The breakpoint is activated.</p> <p><input type="checkbox"/> The breakpoint is not activated. To activate later, click the button  in the view Breakpoints.</p>
--------------------------------------	--

Condition tab

The dialog defines the conditions under which the program execution should stop at the breakpoint.




Tasks

<p>Only break if the breakpoint is hit in one of the following tasks</p>	<p><input checked="" type="checkbox"/> :TwinCAT only evaluates the breakpoint if it is reached by certain tasks. The required tasks must be activated.</p> <p>For example, you can define a single "debug task" and thus prevent other tasks that also use the function block from being affected during debugging.</p>
--	---

Hit Count

Hit Count	<p>Break always: The program always stops at this breakpoint.</p> <p>Alternative: The program stops at the breakpoint when the breakpoint is hit as often as defined below (enter the desired number of hits or select from the number list):</p> <ul style="list-style-type: none"> • Break if the Hit Count matches • Break if the Hit Count is a multiple of • Break if the Hit Count is greater or equal
-----------	---

Condition

Break, if TRUE	<p>Definition of conditional breakpoints. The condition can only be entered in online mode.</p> <p> : TwinCAT evaluates the specified condition and stops the program at this breakpoint if the result is TRUE. Valid Boolean expressions can be entered as a condition. Examples: $x > 100$, $x[y]=z$, $a \text{ AND } b$, boolVar.</p>
----------------	---



The use of conditional breakpoints slows the code execution, even if the condition is not TRUE.

Location tab

Breakpoint Properties ✕

Condition | **Location** | Execution point settings

Location

POU: Simulation [TwinCAT_Project6: PLC: Project6] ▼

Position: Line 8, Column 9 (Impl) ▼

Instances

Instances selected: 0

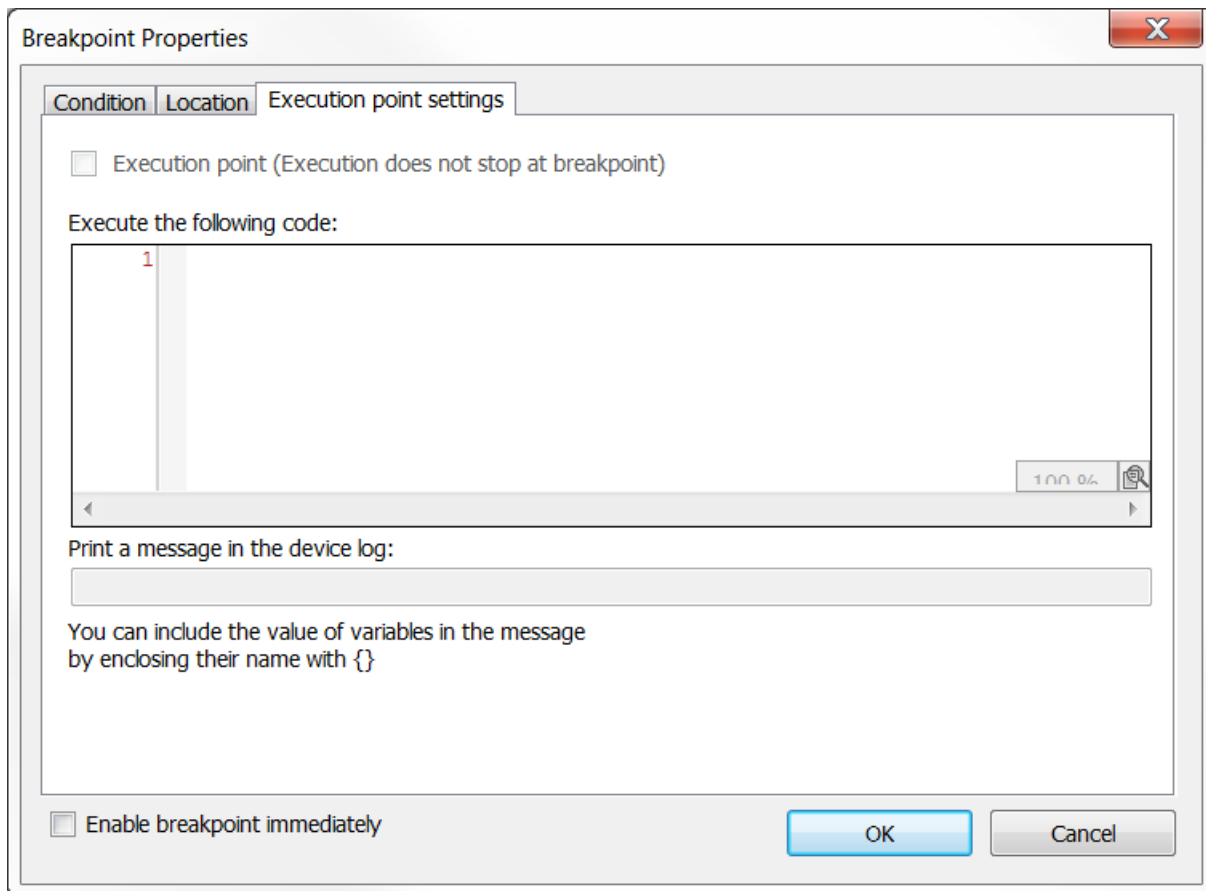
Enable breakpoint immediately

OK
Cancel

POU	Function block of the active PLC project in which the breakpoint is to be positioned.
Position	Position of the breakpoint in the POU. Specification in the form of line and column numbers (text editor) or as network or item numbers.
Instances	<p>For function blocks, you must specify whether the breakpoint should be set in the implementation or in an instance</p> <p><input checked="" type="checkbox"/> TwinCAT sets the breakpoint in the instance. With this option, you select the instance path.</p> <p><input type="checkbox"/> TwinCAT sets the breakpoint in the implementation.</p>

Execution point settings tab

Here an existing breakpoint can be converted into an execution point.




Execution point (Execution does not stop at breakpoint)	<p><input checked="" type="checkbox"/> : The breakpoint becomes the execution point. The execution does not stop at this point, but the specified code is executed.</p> <p>enabled: ● , disabled: ○</p>
Execute the following code	Code that is executed when the execution point is reached. Loop constructs (For, While) and IF or CASE expressions are not possible.
Print a message in the device log	This option is not available.

See also:

- PLC documentation: [Use of breakpoints \[► 211\]](#)

17.6.2 Command Edit Breakpoint

Symbol: 

Function: The command opens the **Breakpoint Properties** dialog.


Call: **Debug** menu, button  in the **Breakpoints** view (**PLC > Window > Breakpoints**)

Requirement: The PLC project is in online mode. The cursor is on a breakpoint.

See also:

- Command New Breakpoint > [Breakpoint Properties dialog \[► 931\]](#)
- PLC documentation: [Use of breakpoints \[► 211\]](#)

17.6.3 Command Enable Breakpoint

Symbol: 

Function: The command enables a disabled breakpoint.


Call: **Debug** menu, button  in the **Breakpoints** view (**PLC > Window > Breakpoints**)

Requirement: The PLC project is in online mode. The cursor is on a disabled breakpoint.

See also:

- PLC documentation: [Use of breakpoints \[► 211\]](#)

17.6.4 Command Disable Breakpoint

Symbol: 

Function: The command disables an enabled breakpoint.

Call: **Debug** menu, button  in the **Breakpoints** view (**PLC > Window > Breakpoints**)

Requirement: The PLC project is in online mode. The cursor is on an enabled breakpoint.

See also:

- PLC documentation: [Use of breakpoints \[► 211\]](#)

17.6.5 Command Toggle Breakpoint

Hotkey: **[F9]**

Function: The command sets a breakpoint or removes an existing breakpoint.


Call: **Debug** menu

Requirement: The PLC project is in online mode. The cursor is on a breakpoint.

See also:

- PLC documentation: [Use of breakpoints \[► 211\]](#)

17.6.6 Command Step over

Symbol: 

Shortcut: **[F10]**

Function: The command executes the instruction the program is currently at and stops before the next instruction in the programming block.

Call: **Debug** menu, **TwinCAT PLC toolbar options**

Requirement: The PLC project is in online mode. The program is at a stop position (debug mode).


If the instruction to be executed contains a call (from a program, function block instance, function, method, or action), the subordinate programming block is completely traversed in one step and the program returns to the call position. Stops before the next statement (in the next line of code).

Select the command **Single step** to jump to a subordinate function block and execute it step by step.

See also:

- [Command Step into \[► 935\]](#)
- PLC documentation: [Stepwise processing of the program \(stepping\) \[► 213\]](#)

17.6.7 Command Step into

Symbol: 

Shortcut: **[F11]**

Function: The command executes the instruction the program is currently at and stops before the next instruction.

Call: **Debug** menu, **TwinCAT PLC toolbar options**

Requirement: The PLC project is in online mode. The program is at a stop position (debug mode).


If the statement to be executed contains a call (from a program, a function block instance, a function, a method or an action), the program jumps to this subordinate programming block. Its code appears in a separate editor. The first instruction there is executed and the program is stopped before the next instruction. The new current stop position is then in the called programming block.

Select the command **Step over**, to stay in the currently active programming block and run through the call in one step.

See also:

- [Command Step over \[► 935\]](#)
- PLC documentation: [Stepwise processing of the program \(stepping\) \[► 213\]](#)

17.6.8 Command Step out

Symbol: 

Shortcut: **[Shift] + [F11]**

Function: The command executes the program until the next return and then stops.

Call: **Debug** menu, **TwinCAT PLC toolbar options**

Requirement: The PLC project is in online mode. The program is at a stop position (debug mode).

If the current stop position is in a subordinate programming block, this is run through to the end. Then the program jumps back to the call position in the calling programming block and stops there (in the line with the call).

If the current stop position is in the main program, the programming block is run through to the end. Then the program jumps back to the beginning (to the program start at the first code line in the programming block) and stops there.

See also:

- PLC documentation: [Stepwise processing of the program \(stepping\)](#) [▶ 213]

17.6.9 Command Run To Cursor

Symbol: 

Function: The command executes a program up to a position marked with the cursor.

Call: Context menu


Requirement: The PLC project is in online mode. The program is at a stop position (debug mode). You have marked any line of code in any programming block with the cursor.

The instructions that lie between the current stop position and the cursor position are executed in one step. Then the execution stops at the cursor position, which thus becomes the next stop position. Note that the line of code where you placed the cursor is reached but not executed.

See also:

- PLC documentation: [Stepwise processing of the program \(stepping\)](#) [▶ 213]


17.6.10 Command Show Next Statement

Symbol: 

Function: The command displays the program statement that will be executed in the next step.

Call: Debug menu

Requirement: The PLC project is in online mode. The program is at a stop position. The hold position is in a line of code that is not visible to you.

The command causes the window with the current stop position, which is yellow in the code and marked with the symbol  , to become active and the stop position to become visible. This is useful if you have many editors open and the hold position is hidden in an editor that is not active.

See also:

- PLC documentation: [Stepwise processing of the program \(stepping\)](#) [▶ 213]

17.6.11 Command Set next statement

Symbol: 

Function: The command determines which statement is executed next.

Call: Context menu

Requirement: The PLC project is in online mode. The program is at a stop position (debug mode). You have marked any line of code in any programming block with the cursor


The line of code marked with cursor becomes the current stop position without executing the statement that was jumped to or the intermediate statements.

See also:

- PLC documentation: [Stepwise processing of the program \(stepping\)](#) [▶ 213]

17.7 TwinCAT

17.7.1 Command Activate configuration

Symbol: 

Function: The command enables a new configuration. The previous old configuration will be overwritten.


Call: Menu **TwinCAT**, **TwinCAT XAE Base toolbar options**

Within the confirmation window that appears after this command is executed, you can set whether the **Autostart boot project** setting should be activated for all PLC projects in the TwinCAT project.

See also:

- Command Activate boot project
- Command Autostart boot project


17.7.2 Command Restart TwinCAT System

Symbol: 

Function: The command starts TwinCAT in Run mode.

Call: Menu **TwinCAT**, **TwinCAT XAE Base Toolbar Options**


17.7.3 Command Restart TwinCAT (Config mode)

Symbol: 

Function: The command starts TwinCAT in configuration mode (Config mode).

Call: Menu **TwinCAT**, **TwinCAT XAE Base Toolbar Options**

17.7.4 Command Reload Devices

Symbol: 

Function: The command loads the created I/O devices.

Call: Menu **TwinCAT**, **TwinCAT XAE Base Toolbar Options**

17.7.5 Command Scan


Symbol: 

Function: The command starts a device scan. The system searches for available I/O devices, connected "boxes" and, if applicable, bus modules and IP-Link extension modules.

Call: Menu **TwinCAT**, **TwinCAT XAE Base Toolbar Options**

Requirement: The "I/O" object is selected in the TwinCAT project structure in the **Solution Explorer**.

17.7.6 Command Toggle Free Run State

Symbol: 

Function: The command sets found I/O devices to free-run mode. This means, for example, that Bus Terminals can set (write) I/O channels to a certain status, without a PLC project or another triggering task being active.


Call: Menu **TwinCAT, TwinCAT XAE Base Toolbar Options**

Requirement: The system is currently in configuration mode.



If the target system was previously in Run mode, the command **Reload Devices** must be executed once before the I/O drivers for the device can be set to free-run state.

17.7.7 Command Show Online Data

Symbol: 

Function: The command connects to the selected target system and displays the parameter values and settings that are active on the target system in the corresponding views.

Call: Menu **TwinCAT, TwinCAT XAE Base Toolbar Options**


17.7.8 Command Choose Target System

Function: Drop-down list to select the target device for the control application.

Call: **TwinCAT XAE Base Toolbar Options**

Select <Local> to load the control code directly into the local runtime of your programming device. If you want to select another target device, select **Choose Target System** from the drop-down list.

17.7.9 Command Show Sub Items

Symbol: 

Function: The command displays the subelements of an element with their properties and values in the overview view of a device. The command can be enabled or disabled. The command does not refer to the representation of the elements in the TwinCAT project tree.

Call: Menu **TwinCAT, TwinCAT Base XAE toolbar options**

Name	Online	Type	Size	>Addr...	In/Out	User ID	Linked to
InfoData							
ChangeCount							
DevId							
AmsNetId							
CfgSlaveCount							
Term 1 (EK1100)							
InfoData							
Term 2 (EL6021)							
Mappings							
Status		Status_E2F...	2.0	26.0	Input	0	
Transmit accepted		BIT	0.1	26.0	Input	0	
Receive request		BIT	0.1	26.1	Input	0	
Init accepted		BIT	0.1	26.2	Input	0	
Buffer full		BIT	0.1	26.3	Input	0	
Parity error		BIT	0.1	26.4	Input	0	
Framing error		BIT	0.1	26.5	Input	0	
Overrun error		BIT	0.1	26.6	Input	0	
Input length		USINT	1.0	27.0	Input	0	
Data In 0		USINT	1.0	28.0	Input	0	
Data In 1		USINT	1.0	29.0	Input	0	
Data In 2		USINT	1.0	30.0	Input	0	
Data In 3		USINT	1.0	31.0	Input	0	
Data In 4		USINT	1.0	32.0	Input	0	
Data In 5		USINT	1.0	33.0	Input	0	
Data In 6		USINT	1.0	34.0	Input	0	
Data In 7		USINT	1.0	35.0	Input	0	
Data In 8		USINT	1.0	36.0	Input	0	
Data In 9		USINT	1.0	37.0	Input	0	

17.7.10 Command Software Protection

Symbol:

Function: The command opens the **Software Protection** dialog.

Call: TwinCAT menu

In the Software Protection dialog you can define the security and user settings for a TwinCAT project.

Further information about the security and user settings can be found in the Software Protection documentation.

17.7.11 Command Hide Disabled Items

Symbol:

Function: This command makes it possible to make disabled objects invisible and visible again in the entire project tree. In this way, the display can be limited to active objects, thereby increasing clarity within the project tree.

Call: Menu TwinCAT, TwinCAT XAE Base toolbar options

17.8 PLC

17.8.1 Window

17.8.1.1 Command Watch List <n>

Symbol:


Function: The command opens the view **Watch List <n>**. You can fill a Watch List with variables from your project in order to be able to monitor, force or write the values for these variables in online mode within a single view. n can be 1, 2, 3, 4, which means that you can configure up to four Watch Lists.

Call: Menu PLC > Window

See also:

- PLC documentation: [Using Watch Lists \[► 226\]](#)

17.8.1.2 Command Watch all forces

Symbol: 

Function: The command opens the **Watch all forces** view, which is a special form of a watch list.

Call: PLC > Window menu

Requirement: A PLC project is open in offline or online mode.

The view contains all variables of the PLC project currently prepared for forcing and all forced variables of the PLC project in a list. In the list you can perform the actions that are also possible in other Watch Lists.

Show all Forces

Tabular display of all variables of the application that have been forced or prepared for forcing

Expression	Variable name
Data type	Data type of the variable
Value	Current forced value of the variable
Prepared value	Value prepared for forcing
Overwritten value at the beginning of the cycle	For inputs, the actual value is already overwritten by the force value before the user code is executed. Thus, this is the actual value. For outputs, this is the forced value.
Overwritten value at the end of the cycle	For outputs, this is the value calculated in the cycle. However, this value is overwritten by the force value at the end of the cycle. For inputs, this is the forced value.

In addition, the **Unforce** selection menu contains the following commands:

- **Unforce and keep all selected values:** For all selected entries in the list, the variables are set to the forced value and forcing is canceled.
- **Unforce and restore all selected values:** For all selected entries in the list, the variables are reset to the value they had before forcing, and forcing is canceled.

See also:

- PLC documentation: [Forcing and writing variables \[► 214\]](#)
- PLC documentation: [Using Watch Lists \[► 226\]](#)

17.8.1.3 Command Cross Reference List

Symbol: 

Function: The command opens the **Cross Reference List** view.

Call: Menu PLC > Window

Cross Reference List view

The view shows a list of cross-references in the project for a symbol. The symbol can be a variable, a POU (program, function block, function) or a user-specific data type (DUT).

The cross reference list basically offers two types of search:

- Text search: Entering a symbol name displays the cross-references of all symbols in the project with this name. If several symbols with the same name are found, the display can be restricted to individual declarations via the context menu.
- Declaration search: The symbol can be selected via the input assistant or by entering a qualified path, for example MAIN.nVar. After that, only the places of use of this symbol are displayed, even if other symbols with the same name exist.

The screenshot shows a window titled 'Cross Reference List' with a search bar containing 'method'. Below the search bar is a table with the following columns: Symbol, POU, Variable, Access, Context, Type, Address, Location, Object, and Comment. The table contains three rows of data:

Symbol	POU	Variable	Access	Context	Type	Address	Location	Object	Comment
bBusy	FB_Sample	bBusy	Declaration	bBusy : BOOL;	BOOL		Line 7 (Ded)	FB_Sample [TwinCAT_Project1: PLC: PLC 1]	
bBusy	FB_Sample.Method1	bBusy	Write	bBusy := TRUE;	BOOL		Line 2, Column 1 (Impl)	Method1 [TwinCAT_Project1: PLC: PLC1: FB_Sample]	
bBusy	FB_Sample.Method2	bBusy	Read	Method2 := bBusy ;	BOOL		Line 1, Column 12 (Impl)	Method2 [TwinCAT_Project1: PLC: PLC1: FB_Sample]	

Toolbar











Name (input field)	<p>Symbol name (variable name, function block name, DUT name)</p> <p>Possible entries:</p> <ul style="list-style-type: none"> • Selection of a declared symbol via the input assistant (button ) • Manual input of the symbol name. <p>Trigger the search via the button  or the [Enter] key. You can use the placeholders "*" (any number of characters) or "?" (exactly any character) in combination with a substring of a variable identifier for the text search. Use "%" if you want to search for IEC addresses. Examples: "%MW8", "%M*"</p> <p>Additional options from outside the view Cross Reference List:</p> <ul style="list-style-type: none"> • Using the context menu command Find All References, if the name of a declared symbol is selected in an editor or the cursor is in the name. • Automatically, if the name of a declared symbol is selected in an editor or the cursor is in the name. An automatic search is also possible if the object is selected in the project tree. Requirements: The Cross Reference List view is open and the TwinCAT option Automatically list selection in cross reference view, category Smart Coding, is activated. <p>The following entries are valid:</p> <ul style="list-style-type: none"> • Variable name, simple or qualified: e.g. "nVar", "MAIN.nVar" • Function block name: e.g. "MAIN", "FB_MyFB" • DUT name: e.g. "ST_MySTRUCT" • Strings in combination with placeholder "*" (any character) or "?" (exactly any character): Example: "nVar*" matches nVar1, nVarGlob2, nVar45 etc.. "nVar?" matches nVar1, nVar2, nVarX etc., but not nVarGlob2, nVar45 etc. • "%<IEC address>": TwinCAT searches for variables assigned to this address and direct memory accesses. Example: "%QB0", "%Q0 := 2" <p>Upper/lower case and spaces at the beginning and end of the input string are ignored.</p>
	Open input assistant to select a symbol.
	Find cross-references: The search is performed.
	Define columns in which to search for the string
Filter (input field)	<p>String to be searched for in the selected columns</p> <p>The found locations are marked in yellow. Cross-references without this string are hidden.</p>
	Show the source position of the previous cross-reference
	Show the source position of the next cross-reference
	<p>Restrict results to current declaration</p> <p>Available when multiple declarations are found for a symbol. Limits the display to the declaration you have just selected in the list.</p>
	Show source position of the selected cross-reference: The focus jumps to the symbol's usage location.
	Print Cross Reference List: The standard dialog for setting up a print job appears.

Table of cross-references found

Symbol	The locations for the symbols (variables, POU, DUTs) are grouped according to their declaration. The declaration location forms the root node, the usage points in the project appear indented below it. The system displays the exact expression, which the symbol has at the point of use. Example: If there is a global variable "nVar" in the project and a locally declared variable "nVar" in a function block, then after a text search for the cross-references two root node entries appear in the list, and the respective point of use of variable "nVar" is shown below.
Function block	Function block name, DUT name; also task name in case of a function block call in the task configuration, for example.
Variable	Pure variable name. Example: "nVar".
Access	Type of access to the variable at the point of use: Declaration / Read / Write / Call. Special case for pointers: An assignment of the type <code>pSample := ADR(nVar1)</code> is displayed as <code>Write Address</code> when searching for "nVar1". Reason: Any write accesses to "pSample" are not displayed when searching for "nVar1". Write accesses are also possible via the pointer variable.
Context	Context of the use of the variables. Example: "nVar := 1"
Type	Data type of the variable.
Address	IEC address, if assigned to the variable. Example "AT%QB0"
Position	Position of the usage location within the editor of the POU concerned: for example, line number, network number, declaration part, or implementation part. Example: "Line 1, column 1 (Impl)" or "Line 9 (Decl)".
Object	POU name + in square brackets the complete path of the point of use. Example: "MAIN [TwinCAT_SampleProject: PLC: SamplePLCProject]"
Comment	Comment, if present in the variable declaration.

The search returns all locations in the project as well as in inserted, uncompiled libraries.


Commands in the context menu of the Cross Reference List

Display source code position	Opens the affected function block and marks the usage location: for root entries the declaration, for child entries below it the respective usage location. Alternatively, you can double-click on a line.
Restrict results to current declaration	Limits the display of results to the selected symbol declaration in case of multiple found declarations.
Expand All	The list shows all the individual locations.
Collapse All	The list only shows the root nodes of all locations.

See also:

- Command 'Limit results to current declaration'
- Command Collapse All Folds
- Command Expand All Folds
- PLC documentation: [Using the Cross Reference List to find Occurrences \[► 157\]](#)

17.8.1.4 Command Breakpoints

Symbol: 

Function: The command opens the **Breakpoints** view.

Call: Menu **PLC > Window**

Breakpoints view









The view provides an overview of all defined breakpoints of an application. All breakpoint commands are available within the view.

POU	Location	Instance path	Tasks	Condition	Hit count condition	Current hit count	Watched values last updated
MAIN	Line 1, Column 1 (Impl)	TwinCAT_Device.Project1.MAIN	(any)	Break always	Break always	0	

Table of current breakpoints

Application	Select the desired PLC project from the list.
POU	Name of the function block containing the breakpoint.
Position	Breakpoint position within the POU <ul style="list-style-type: none"> Text Editor: Row and column number Graphic editor: Network or element number "(Impl)" in the case of function blocks indicates that the breakpoint is in the implementation of the function block, not in an instance.
Instance path	Complete object path of the breakpoint position.
Tasks	Name of the tasks for whose execution the breakpoint is to be effective. If there is no restriction, it says "(all)".
Condition	<ul style="list-style-type: none"> Break always: No additional activation condition defined. The breakpoint is always active. Boolean expression. The expression must return TRUE for the breakpoint to be active.
Hit count condition	Indicates when (in which dependency on the hit count) the breakpoint should become effective.
Current hit count	Specifies how often the breakpoint has already been passed through ("hit") during execution.


Toolbar

	New Breakpoint (corresponds to the command Command New Breakpoint [▶ 930] in the Debug menu)	Opens the Breakpoint Properties dialog
	Clear breakpoint	Removes the breakpoint. Do not confuse this command with the Disable command.
	Enable/disable breakpoint (corresponds to the commands Command Enable Breakpoint [▶ 934] and Command Disable Breakpoint [▶ 934] in the Debug menu)	Switches the breakpoint or execution point between "enabled" and "disabled" status. <ul style="list-style-type: none"> ● breakpoint enabled ○ breakpoint disabled ● Execution point enabled ○ Execution point disabled In contrast to Clear breakpoint , a disabled breakpoint remains in the list and can be enabled again.
	Properties	Opens the Breakpoint Properties dialog for editing the breakpoint parameters. In online mode, you can convert the breakpoint to a execution point here.
	Go to source code position	Opens the online view of the respective function block. The cursor is at the breakpoint position.
	Delete All Breakpoints	Deletes all breakpoints and execution points of the application. The list is emptied. Not to be confused with disabling!
	Enable all breakpoints	Enables all currently disabled breakpoints and execution points.
	Disable all breakpoints	Disables all currently enabled breakpoints and execution points. The points remain in the list and can be re-enabled.

See also:

- [Command New Breakpoint > Breakpoint Properties dialog \[▶ 931\]](#)
- [Command Toggle Breakpoint \[▶ 934\]](#)
- PLC documentation: [Use of breakpoints \[▶ 211\]](#)

17.8.1.5 Command Call Stack

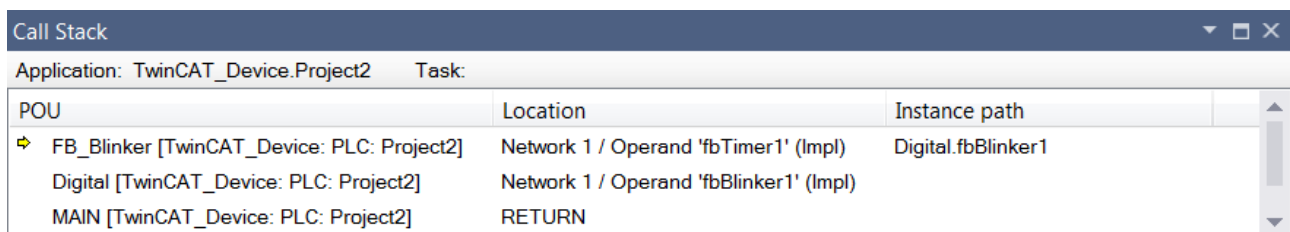
Symbol: 

Function: The command opens the **Call Stack** view.

Call: Menu **PLC > Window**

Callstack view

This view is useful if you want to run programs step-by-step. It shows the currently reached position with the complete call path.



Application	Name of the active PLC project that controls the currently reached program block.
Task	Name of the task that controls the currently reached program block.
POU	Name of the program block in which the program execution is located. The first line in the list describes the current execution position. It is marked with a yellow arrow. If this position is in a function block called by another function block, the position of the call is described in the second line. If the caller is called by another function block, this call position is described in the third line and so on.
Location	Position within the program block where the program execution is located <ul style="list-style-type: none"> • Row and column number for text editors • Network or element number for graphical editors
Instance path	Instance in which the program is executed.

The Call Stack is also available in offline mode and in normal online mode if you are not using any debugging functions. In this case, it contains the last position displayed during a step-by-step execution, but in "gray" font.



In contrast to the **Call Stack** view, the **Call Tree** view provides call information on a function block at any time.

See also:

- PLC documentation: [Use of breakpoints \[► 211\]](#)

17.8.1.6 Command Memory

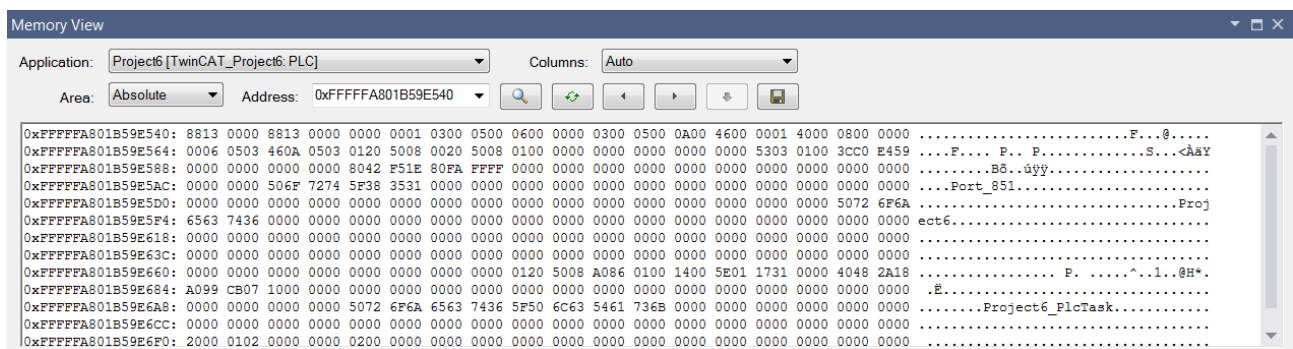
Symbol:







Function: The command opens the **Memory View**.

Call: Menu **PLC > Memory**

Requirement: As a rule, the control system supports the functionality. At least one application is loaded and in online mode.

Memory View



Application	Select the PLC project for which the memory view is to be displayed. You must be logged on to the controller with this project. It does not have to be the "active PLC project".
Area	<ul style="list-style-type: none"> • Absolute: Memory is addressed directly and completely. The address is in the input field next to it. • Area <i>i</i>: Memory areas of the controller, starting with Area 0. Memory areas reserved exclusively for code are not displayed.
Address	Absolute start address of the core dump Requirement: Absolute is selected in the range.
Offset	Address offset to the selected memory area in bytes, for example 0x0200, 16#0200 or as decimal number 512 Requirement: A memory area is selected in area, e.g. Area 0. TwinCAT offers all currently used memory areas for selection. Memory areas reserved exclusively for code are not displayed.
	Find out the address for a variable: Input Assistant for selecting an IEC variable appears. If you have selected a variable, TwinCAT presets the start address with the variable address.
	Load/update memory view
	Show previous segment: Navigate to the previous memory segment
	Show next segment: Navigate to the next memory segment
	Notice TwinCAT does not check whether the changes are permissible. You can crash the application by making inadvertent changes Load changes to PLC: TwinCAT transfers the new data to the controller. Requirement: You have overwritten one or more bytes in the memory view.
	Save the contents of the memory to a file: The dialog "Memory content as binary file" appears. Select a location.
Columns	Number of columns in the hexadecimal representation of the memory dump. Two bytes are displayed per column. With Auto, the number of columns adjusts to the window size. To the right of it, the data is displayed as text.

17.8.1.7 Command Call Tree

Symbol: 

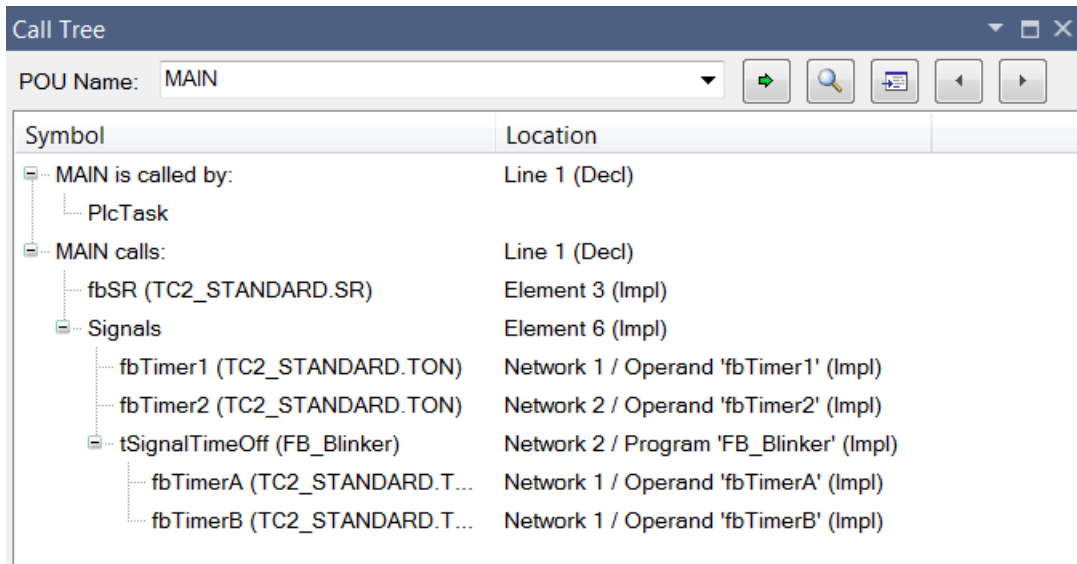
Function: The command opens the **Call Tree** view.


Call: Menu **PLC > Window**

Call Tree view




The Call Tree is available at any time before the application is compiled. It is a static representation of the callers and calls of the function block, which you can specify explicitly. This means that the tree always contains two root nodes, under which the respective call sequence is displayed as consecutive indented entries. Recursive calls are quickly recognizable in this tree view.

Example of a Call Tree (1) for function block (2) MAIN:



POU Name	The name of the program block can be entered manually, by dragging it from another view, or by using the  button. The selection list contains the last function block names entered.
----------	--

Toolbar and keyboard operation

 Find function block	TwinCAT searches for the function block specified in "Function block name" and displays its callers and calls.
 Take function block from Input Assistant	The Input Assistant dialog appears for selecting a function block call or instance call. The Call Tree is automatically updated after the selection.
 Display source code position of the selected function block	TwinCAT jumps to the point where the function block is used in the source code of your program.
<input type="checkbox"/> Show source code position of the next function block	The selection in the Call Tree jumps to the next or previous function block in the call structure. At the same time, the corresponding source code position is opened in the respective editor. Double-clicking on an entry in the Call Tree also opens the corresponding source code position.
<input type="checkbox"/> Show source code position of the previous function block	

Display of the Call Tree

Location	For the root nodes in the Call Tree: Line number of the declaration ("Decl") of the function block. For the callers or calls under the root node: Depending on the implementation language, line number, column number, network number of your position.
----------	---

Context menu for the entry currently selected in the tree

Collapse All	The expanded entries in the Call Tree are collapsed except for the two root nodes.
Display source code position	TwinCAT jumps to the point where the function block is used in the source code of your program.
Set as new root node	The entry selected in the Call Tree appears in "function block name". The tree is automatically adapted for the new root nodes.



In contrast to the static Call Tree, which always provides call information for a function block, the **Call Stack** view is intended for immediate information during the step-by-step processing of a program. The Call Stack always shows the complete call path of the currently reached position.

17.8.1.8 Command Online Change Memory Reserve Settings

Function: The command opens the **Online Change Memory Reserve** view.

Call: Menu **PLC > Window**.

This view is used to configure the memory reserves for the online change for function blocks.

Browse application	<ul style="list-style-type: none"> • Searches the selected PLC project for function blocks and displays them in the Function blocks area • Updates the Function blocks area after the PLC project has been compiled again • Updates the Function blocks area after an online change
Selection list with the PLC projects of the open TwinCAT project	Selection of the PLC project whose function blocks are to be displayed and/or edited in this view

Function blocks:

All	All function blocks of the selected PLC project are displayed.
No memory reserve	All function blocks with a memory reserve of 0 bytes are displayed.
<Memory reserve> bytes	Display of all function blocks with the number of bytes defined in memory reserve .
Information on the function blocks Multiple selection is also possible when selecting a function block for configuring the memory reserve.	
Function block	Name of the function block
Size	Size of the function block Size of an instance of the function block Specification in bytes
Number of instances	Number of instances of the function block in the project
Memory reserve	Displays the memory reserve for each instance of the function block
Additional memory for all instances	Product of number of instances and memory reserve
Remaining memory reserve	Number of bytes that are available as reserve per function block instance

Settings:

Memory reserve (in bytes)	Input field for the memory reserve for the selected function block. Specification in bytes Requirement: The PLC project is not yet on the controller or you have allowed the memory reserve to be changed by clicking the Allow button in the Allow editing area.
Apply to selection	The memory reserve (in bytes) is assigned to the function block, and the Memory reserve table column is updated. If multiple function blocks are selected, the entered value is assigned to each function block. To update the columns Size , Number of instances , Additional memory for all instances , and Remaining memory reserve size , first choose Create > Create , then click the Browse application button.

Enable editing:

Enable	The input field Memory reserve (in bytes) becomes editable. This button is changed to Editable .
--------	---




Information:

Number of FBs	Total number of function blocks in the PLC project
Additional memory for all instances	Sum of the memory reserves of all function block instances of the PLC project Specification in bytes

See also:

- PLC documentation: Program PLC project > [Configure memory reserve for online change](#) |▶ 133]

17.8.1.9 Command PLC BookmarksSymbol: **Function:** The command opens the view **Bookmarks**.**Call:** **PLC > Window** menu

 Previous Bookmark	Jumps to the bookmark displayed in the table one line above the selected line and opens the corresponding POU in the editor.
 Next Bookmark	Jumps to the bookmark displayed in the table one line below the selected line and opens the corresponding POU in the editor.
	Deletes the selected bookmark from the table and in the corresponding POU.

Listing of the project's bookmarks with the information bookmark, object and position:

Bookmark	Designation of the bookmarks assigned by TwinCAT in numbered, ascending order: Bookmark_0, Bookmark_2 etc. When the bookmark is selected and you click in the field, it becomes editable and you can change the bookmark name.
Object	Name and project path of the POU in which the bookmark is set
Position	Position of the bookmark within the POU Example: Line 3, Column 1 (Impl) (Impl): in the implementation part of the POU (Decl): in the declaration part of the POU

You can change the order of bookmarks by drag and drop.

If you double-click a line, TwinCAT opens the corresponding object in the editor and jumps to this bookmark.

See also

- [Command Previous Bookmark \[► 954\]](#)
- [Command Next Bookmark \[► 953\]](#)
- [Set and use bookmarks \[► 158\]](#) ("PLC" documentation)

17.8.2 Core dump

17.8.2.1 Command Generate core dump

Function: The command causes TwinCAT to first check whether a core dump file is already available on the target system.

- If a core dump file is available on the target system, TwinCAT offers you to load this file into the project directory. There are three different ways to respond to the query as to whether the core dump file is to be loaded from the target system.
 - Yes: If the core dump file of the target system matches the currently logged-in PLC project, TwinCAT loads the core dump file from the target system to the project directory. You can open this file by subsequently logging out the PLC project and using the [Command Load core dump \[► 952\]](#).
 - No: A new core dump file will be generated in the project directory. The prerequisite for this is that the PLC project is currently at a breakpoint or that an exception error has occurred.
 - Cancel: The generation of a core dump file is aborted.
- If no core dump file is available on the target system, TwinCAT initiates the generation of a new file with the current PLC project data in the project directory. The prerequisite for this is that the PLC project is currently at a breakpoint or that an exception error has occurred.

The core dump file generated is saved directly in the PLC project directory:

<PLC_project_name>.<PLC_project_GUID>.core

Call: menu PLC > Core Dump

Requirement: The PLC project is in online mode.

● Automatic generation of a core dump on the target system

i If the PLC project running on a target system is not currently logged into a development environment, the runtime system automatically generates a core dump on the target system in the event of an exception error. This file is located by default in the boot folder of the target system (by default under < TC3.1.4026.0: C:\TwinCAT\3.1\Boot\Plc; >=TC3.1.4026.0: C:\ProgramData\Beckhoff\TwinCAT\3.1\Boot\Plc). If desired, the storage path of the automatic core dump creation can also be modified, see [Error analysis with core dump \[► 247\]](#) for more information.

The automatic loading of this core dump file from the target system to the local project directory is possible with the help of the **command Generate core dump**. It is also possible to copy the core dump file from the target system to the development computer.

Displaying the dump using the [command Load core dump \[► 952\]](#) can thus be used for (subsequent) error analysis.

● Core dump can only be used with the associated compile info file

i If you archive or save a core dump file, please note that the associated project and associated compile info file (*.compileinfo file, which is stored, for example when creating the project, in the "_CompileInfo" folder) must be present in order to load a core dump. If this is not the case, TwinCAT cannot use the dump later.

Please also note here the setting options on the [Settings tab \[► 926\]](#). With the help of the **Core Dump** setting, you can configure whether the core dump file, which may be located in the project directory, is to be saved together with the available compile info files in a TwinCAT file archive.

See also:

- PLC documentation: PLC project at runtime > [Error analysis with core dump \[► 247\]](#)
- [Command Load core dump \[► 952\]](#)

17.8.2.2 Command Load core dump

Function: TwinCAT searches the project directory for core dump files.

- If TwinCAT finds a core dump file in the project directory, you are asked whether you want to load this core dump or browse for a dump file.
- If TwinCAT does not find a core dump file in the project directory, you can browse for another dump file.

Loading into the project causes an online view of the PLC project to appear, with the state the PLC project had at the time when the core dump was created. You can view the variable values contained in it. The call tree is also available.

Call: Menu PLC > Core dump

Requirement: The application is in offline mode.

i The Core Dump view can only be closed again using the [Command Close core dump \[► 953\]](#) command. The Logout command is not effective in this view.

● Core dump usable only with associated compile info file

i If you archive or save a core dump file, please note that the associated project and associated compile info file (*.compileinfo file, which is stored, for example when creating the project, in the "_CompileInfo" folder) must be present in order to load a core dump. If this is not the case, TwinCAT cannot use the dump later.

Please also note here the setting options on the [Settings tab \[► 926\]](#). With the help of the **Core Dump** setting, you can configure whether the core dump file, which may be located in the project directory, is to be saved together with the available compile info files in a TwinCAT file archive.

See also:

- PLC documentation: PLC project at runtime > [Error analysis with core dump \[► 247\]](#)
- [Command Generate core dump \[► 951\]](#)
- [Command Close core dump \[► 953\]](#)

17.8.2.3 Command Close core dump

Function: The command closes the core dump view of the PLC project that is currently open in the development environment.

Call: Menu PLC > Core dump


Requirement: The PLC project is in offline mode and you have loaded a core dump file into the project.

See also:

- PLC documentation: PLC project at runtime > [Error analysis with core dump \[► 247\]](#)

17.8.3 PLC Bookmarks

17.8.3.1 Command Enable/disable bookmarks

Symbol: 

Function: The command sets or deletes a bookmark at the current position.

Call: PLC > PLC Bookmarks menu

Requirement: A POU is open in the editor and the cursor is in a program line.

See also


- [Bookmark \[► 158\]](#) (Documentation "PLC")

17.8.3.2 Command Next Bookmark

Symbol: 

Function: The command jumps to the next bookmark in the view **Bookmarks** and in the project and opens the corresponding POU. The order in which the bookmarks are jumped to corresponds to the order of the bookmarks in the table of the view **Bookmarks**.

Call:

- Menu PLC > PLC Bookmarks
- Button  **Next bookmark** in the view **Bookmarks**


Requirement:

- A project is open
- The view **Bookmarks** is open

See also:


- [Command PLC Bookmarks \[► 950\]](#)
- [Command Next Bookmark \(active editor\) \[► 954\]](#)
- [Bookmark \[► 158\]](#) (Documentation "PLC")

17.8.3.3 Command Previous Bookmark

Symbol: 

Function: The command jumps to the previous bookmark in the view **Bookmarks** and in the project and opens the corresponding POU. The order in which the bookmarks are jumped to corresponds to the order of the bookmarks in the table of the view **Bookmarks**.

Call:

- Menu **PLC > PLC Bookmarks**
- Button  Previous bookmark in the view Bookmarks


Requirement:

- A project is open
- The view **Bookmarks** is open

See also:

- [Command PLC Bookmarks \[► 950\]](#)
- [Command Previous Bookmark \(active editor\) \[► 954\]](#)
- [Bookmark \[► 158\]](#) (Documentation "PLC")

17.8.3.4 Command Clear All Bookmarks

Symbol: 

Function: The command deletes all bookmarks of the opened project.

Call: **PLC > PLC Bookmarks** menu

Requirement: A POU is open in the editor and the cursor is in the POU.

See also:

- [Command Clear All Bookmarks \(active editor\) \[► 955\]](#)
- [Bookmark \[► 158\]](#) (Documentation "PLC")

17.8.3.5 Command Next Bookmark (active editor)

Symbol: 

Function: The command jumps to the next bookmark in the active editor.


Call: **PLC > PLC Bookmarks** menu

Requirement: A POU is open in the editor and the cursor is in the POU

See also:

- [Command Next Bookmark \[► 953\]](#)
- [Bookmark \[► 158\]](#) (Documentation "PLC")

17.8.3.6 Command Previous Bookmark (active editor)

Symbol: 

Function: The command jumps to the previous bookmark in the active editor.


Call: PLC > PLC Bookmarks menu

Requirement: A POU is open in the editor and the cursor is in the POU

See also:

- [Command Previous Bookmark \[► 954\]](#)
- [Bookmark \[► 158\]](#) (Documentation "PLC")

17.8.3.7 Command Clear All Bookmarks (active editor)

Symbol: 

Function: The command deletes all bookmarks in the active editor

Call: PLC > PLC Bookmarks menu

Requirement: A POU is open in the editor and the cursor is in the POU.

See also:

- [Command Clear All Bookmarks \[► 954\]](#)
- [Bookmark \[► 158\]](#) (Documentation "PLC")

17.8.4 Command Download

Function: The command causes the active PLC project to be compiled and then downloaded to the controller.

Call: PLC menu

Requirement: The PLC project is in online mode.

With this command, TwinCAT performs a syntax check and generates the program code. This code is loaded onto the controller. TwinCAT also generates the compile log <projectname>.<devicename>.<application ID>.compileinfo in the project directory.



During the download, all variables except persistent variables are reinitialized.

The description of the **Login** command explains the possible situations when logging in and loading.

If you try to load a PLC project while the same version of this project is already on the controller, the following message appears: "The program is unchanged. Application was not loaded". TwinCAT does not load the project onto the PLC.

When loading, a log of the actions that have taken place (creating the code, carrying out initializations, etc.) is displayed in the **Output** view. In addition, information about memory areas, the size of the code, global data and allocated memory is displayed. For the sake of clarity, in contrast to online change, the changed function blocks are no longer listed.

See also:

- [Command Login \[► 957\]](#)

17.8.5 Command Online Change

Function: The command is used to initiate an online change to the currently active PLC project. TwinCAT only reloads the modified parts of a PLC project that is already running on the controller into the controller.

Call: PLC menu

Requirement: The PLC project is in online mode.

An online change is not possible after the **Clean All** and **Clean** commands. The cleanup process deletes the compile information (compile log) that is automatically saved each time code is generated, and which forms the basis for an online change.

⚠ WARNING

Damage to property and persons due to unexpected behavior of the machine or system

An online change modifies the running application program and does not cause a restart. Depending on the machine being controlled, the machine or workpieces may be damaged or the health and life of people may be endangered.

- Make sure that the new program code results in the desired behavior of the controlled system.

● **Project-specific initializations**

i When an online change is performed, the project-specific initializations (homing etc.) are not executed because the machine retains its status. For this reason, the new program code may not have the desired effect.

● **Major changes in the download code**

i If the online change causes significant changes in the download code (e.g. shifting of variables is required), a dialog provides information about the effects and allows you to cancel the online change.

● **Fast online change**

i For small changes (e.g. in the implementation section, with no shifting of variables required), a "fast online change" is performed. In this case, only the modified function block is compiled and reloaded. In particular, no initialization code is generated in this case. This also means that no code for initializing variables with the attribute 'init_on_onlchange' is generated. Usually this will not have any effect, since the attribute tends to be used to initialize variables with addresses, but a variable cannot change its address during a fast online change.

To ensure the `init_on_onlchange` attribute is applied to the entire application code, deactivate fast online change for the PLC project using the `no_fast_online_change` compiler definition. To this end, insert the definition in the [Compile category \[► 910\]](#) of the PLC project properties.

● **The attribute 'init_on_onlchange' has no effect for individual FB variables**

i The [Attribute 'init_on_onlchange' \[► 807\]](#) only applies to global variables, program variables and local static variables of function blocks.

To reinitialize a function block during an online change, the function block instance must be declared with the attribute. The attribute is not evaluated for a single variable in a function block.

Pointer variables

Pointers retain their value from the last cycle. If a pointer points to a variable that was resized by the online change and moved in the memory as a result, the pointer variable no longer returns the correct position of the variable. Make sure that pointers are reassigned in each cycle.

During an online change with possibly unintended consequences, TwinCAT lists the changed interfaces, affected variables and all function blocks for which new code has been generated in the Details dialog box. If storage locations change, a dialog is displayed to indicate possible problems with pointers.

See also:

- PLC documentation: [Programming a PLC project \[► 66\]](#)

What prevents an online change?


After certain TwinCAT actions an online change on a controller is no longer possible. After that, a [download \[▶ 253\]](#) of the project is always required. A typical case are the actions **Clean** and **Clean all**, which delete the compilation information stored during the last download. But there are also "normal" editing actions that result in an online change not being possible the next time you log in. The following actions can prevent an online change:

Check functions	Activating or removing a check function [▶ 163] (CheckBounds, CheckRange, CheckDiv etc.). Changing the interface of a check function (including inserting or deleting local variables).
Task configuration	Changing the configuration settings.
Project settings	Category Compile [▶ 910] : Changing the compiler defines in the Settings section (replace constants) Category Common [▶ 906] : Minimize ID changes in TwinCAT files.
Function block properties	Change of option External implementation
Function block	Changing the basic block of a function block (EXTENDS [▶ 191] FB_Base), or inserting or deleting such a basic block. Changing the interface list (IMPLEMENTS [▶ 198] I_Sample). Exception: Adding a new interface at the end of the list.
Data type	Changing the data type of a variable from one user-defined data type to another user-defined data type (e.g. from TON to TOF). Changing the data type from a user-defined data type to a basic data type (e.g. from TON to TIME). Note: As a workaround, always change the name of the variable at the same time as the data type. As a result, the variable is initialized as a new variable, and the old variable is removed. An online change is then possible.

See also:

- [Performing an Online Change \[▶ 251\]](#)
- [Command Login \[▶ 957\]](#)

17.8.6 Command Login

Symbol: 

Function: This command connects the programming system (the selected PLC project) with the target system (controller) and thus establishes online operation. An instance of the PLC project is created on the target system and loaded.

Call: **PLC** menu or **TwinCAT PLC toolbar options** or context menu of the PLC project object (<PLC project name>Project) in the **Solution Explorer**

Requirement: The PLC project is error-free and the target system is in Run mode.

Possible login situations:

- The PLC project does not yet exist on the controller: You will be prompted to confirm the download.
- The PLC project is already on the controller and has not been changed since the last download. Logging in takes place without further interaction with you.
- The PLC project is already on the controller, but has been changed since the last download. You will be prompted to choose one of the following options:
 - Login with Online Change (please refer to the notes in section "[Command Online Change \[▶ 955\]](#)")

- Login with download
- Login without changes

At this point you can also update the boot project on the controller.

- An unknown version of the PLC project is already on the controller. You will be asked whether TwinCAT should replace it.
- A version of the PLC project is already running on the controller. You will be asked whether TwinCAT should nevertheless log in and overwrite the PLC program that is currently running.
- The PLC program on the controller is currently stopped at a breakpoint. You have logged out and changed the program: TwinCAT warns you that, in the event of an online change or download, the PLC will be completely stopped. This happens even if there are several tasks and only one of them is affected by the breakpoint.

Compiling the project before logging in

If a PLC project has not been compiled since its last change, TwinCAT compiles the project before logging in. This operation corresponds to the command **Compile in logged-out state**.

If errors occur during the compilation, a message dialog appears. The errors are displayed in the **Error List** view. You can then decide whether you want to log in without loading the program onto the controller.

See also:

- [Command Build PLC project \[► 929\]](#)

Error on login

If an error occurs while logging on to the controller, TwinCAT interrupts the loading process with an error message. The error dialog allows you to display the error details. If an exception error has occurred and the text ***SOURCEPOSITION*** is contained in the log message, you can use the command **Display in Editor** to display the relevant function in the editor. The cursor jumps to the line causing the error.

Output of information on the loading process

If TwinCAT loads the project onto the controller when logging in, the following information is displayed in the message window:

- Generated code size
- Size of global data
- Resulting memory requirement on the controller
- A list of the affected function blocks (for online change)

i In online mode, you cannot change the settings of devices or modules. To change device parameters, you must log the PLC project out. Depending on the bus system, there may, however, be some special parameters that you can change in online mode.

i TwinCAT stores the view configuration separately for online and offline mode. In addition, views that cannot be used in an operation mode are closed. For this reason, the view may change automatically when you log in.

17.8.7 Command Start

Symbol: 

Keyboard shortcut: **[F5]**


Function: The command starts the execution of the program.

Call: Menu **PLC**, **TwinCAT PLC Toolbar Options**

Requirement: The PLC project is in online mode.

When you call the command from the **PLC** menu, it affects the currently active PLC project.

17.8.8 Command Stop

Symbol: 

Keyboard shortcut: **[Shift] + [F5]**


Function: The command stops the execution of the program.

Call: Menu **PLC, TwinCAT PLC Toolbar Options**

Requirement: The PLC project is in online mode.

When you call the command from the **PLC** menu, it affects the currently active PLC project.

17.8.9 Command Logout

Symbol: 

Function: The command terminates the connection between the development system and the target system (controller or simulated device), thus switching to offline mode.

Call: Menu **PLC, TwinCAT PLC Toolbar Options**

17.8.10 Command Reset cold

Symbol: 

Function: The command resets all variables of the active PLC project to their initialization values, except the PERSISTENT variables and the RETAIN variables.

Call: Menu **PLC, TwinCAT PLC toolbar options**

Requirement: The PLC project is in online mode.

All variables of the active PLC project are reset, with the exception of the PERSISTENT variables and the RETAIN variables. The situation is the same as when an application program that has just been loaded onto the controller ("cold start") is started.

Breakpoints of the PLC program that were enabled before the active PLC project was reset are still enabled after execution of the command. Breakpoints that were previously disabled are still disabled after the command is executed.

If you select the command while the program is stopping at a breakpoint, you will be asked whether the current cycle is to be terminated. Alternatively, TwinCAT carries out the reset immediately. However, not all runtime systems are able to perform a reset without terminating the current cycle.

After the reset, you must start the PLC program with the **Start** command.

See also:

- PLC documentation: [Remanent Variables - PERSISTENT, RETAIN \[► 691\]](#)
- PLC documentation: [Resetting the PLC project \[► 218\]](#)
- [Command Reset origin \[► 959\]](#)
- [Command Start \[► 958\]](#)

17.8.11 Command Reset origin

Symbol: 

Function: The command resets all variables of the active PLC project, including the remanent variables (RETAIN, PERSISTENT variables), to their initialization values and deletes the application program from the controller.

Call: Menu **PLC**, **TwinCAT PLC Toolbar Options**

Requirement: The PLC project is in online mode.

A reset deactivates the currently set breakpoints in the program. If you select the command while the program is stopping at a breakpoint, you will be asked whether the current cycle is to be terminated. Alternatively, TwinCAT carries out the reset immediately. However, not all runtime systems are able to perform a reset without terminating the current cycle.


See also:

- [Command Reset cold \[► 959\]](#)

Also see about this

- [Remanent Variables - PERSISTENT, RETAIN \[► 691\]](#)
- [Resetting the PLC project \[► 218\]](#)

17.8.12 Command Single cycle


Symbol: 

Function: The command executes the active PLC program for one cycle.

Call: Menu **PLC**, **TwinCAT PLC Toolbar Options**

Requirement: The PLC project is in online mode and the program is at a program step.

17.8.13 Command Flow Control

Symbol: 

Function: The command activates or deactivates the flow control.

Call: Menu **PLC**, **TwinCAT PLC Toolbar Options**

Requirement: The PLC project is in online mode.



An active flow control extends the runtime of the PLC project!

See also:

- PLC documentation: [Flow Control \[► 219\]](#)

17.8.14 Command Force values

Symbol: 

Function: The command permanently sets the value of a variable on the controller to a predefined value.

Call: Menu **PLC**, **TwinCAT PLC Toolbar Options**

Requirement: The PLC project is in online mode.

⚠ CAUTION**Material damage and personal injury due to unexpected behavior of the controlled system**

Changing the values of variables in a PLC program running on the controller can lead to unexpected behavior of the controlled system. Depending on the controlled system, damage to equipment and workpieces can occur or people's health and lives may be endangered.

- Evaluate possible risks before forcing variable values and take appropriate safety precautions.

With this command, TwinCAT permanently sets one or more variables of the active application on the controller to defined values. This setting is carried out at the beginning and end of each processing cycle. Order of execution: 1. Read inputs, 2. Force values, 3. Execute code, 4. Force values, 5. Write outputs.

You can prepare values by

- clicking in the **prepared value** field in the declaration part and entering the new value. For a Boolean variable, change the value by single-clicking in the field.
- Click the inline monitoring field in the implementation part of the FBD/LD/IL editor and enter the new value.
- Click in the **prepared value** field in the Monitor window and enter the new value.

A "forced" value is marked with an **F**.

Expression	Type	Value
iCount	INT	45
bSwitich	BOOL	F TRUE
Axis1	AXIS_REF	

TwinCAT performs forcing until it is explicitly cancelled by the user through

- the command **Unforce values**
- cancellation of forcing via the **Prepare Value** dialog
- Logging out of the application



The command **Force values for all applications**, which affects all PLC projects in the TwinCAT project, is not included in a menu by default.

See also:

- [Command Unforce values \[► 962\]](#)
- [Dialog Prepare Value \[► 961\]](#)
- PLC documentation: [Forcing and Writing Variables Values \[► 214\]](#)

17.8.14.1 Dialog Prepare Value

Function: The dialog is used to prepare a value for an already forced variable. TwinCAT executes the prepared action with the next forcing.

Call: TwinCAT opens the dialog in the following situations:

- if you click in the field **prepared value** of a forced variable in the declaration part
- if you click in the inline monitoring field of a forced variable in the implementation part
- if you click in the **prepared value** field of a forced variable in the Monitor window

Preparing a new value for the next write or force operation	Value that TwinCAT writes to the variable during the next force operation
Remove preparation with a value	TwinCAT deletes the prepared value.
Release the force, without modifying the value.	TwinCAT retains the forced value and terminates forcing. TwinCAT marks the variable with <Unforce>.
Release the force and restore the variable to the value it had before forcing it.	TwinCAT resets the forced value and terminates forcing. The variable is marked with <Unforce and restore>.

See also:

- [Command Force values \[► 960\]](#)

17.8.15 Command Unforce values

Symbol:

Function: The command resets the forcing of all variables. The variables get their current value from the controller.

Call: Menu **PLC, TwinCAT PLC Toolbar Options**

Requirement: The PLC project is in online mode.

⚠ CAUTION

Material damage and personal injury due to unexpected behavior of the controlled system

Changing the values of variables in a PLC program running on the controller can lead to unexpected behavior of the controlled system. Depending on the controlled system, damage to equipment and workpieces can occur or people's health and lives may be endangered.

- Evaluate possible risks before resetting forced variable values and take appropriate safety precautions.



The command **Unforce values for all applications**, which affects all PLC projects in the TwinCAT project, is not included in a menu by default.

See also:

- [Command Force values \[► 960\]](#)
- PLC documentation: [Forcing and Writing Variables Values \[► 214\]](#)

17.8.16 Command Write values

Symbol: 

Function: The command once sets the value of a variable on the controller to a predefined value.

Call: Menu **PLC, TwinCAT PLC Toolbar Options**

Requirement: The PLC project is in online mode.

CAUTION

Material damage and personal injury due to unexpected behavior of the controlled system

Changing the values of variables in a PLC program running on the controller can lead to unexpected behavior of the controlled system. Depending on the controlled system, damage to equipment and workpieces can occur or people's health and lives may be endangered.

- Evaluate possible risks before writing variable values and take appropriate safety precautions.

With this command, you set one or more variables of the active PLC project on the controller to defined values once. Writing takes place once at the beginning of the next cycle.

You can prepare values by

- clicking in the **prepared value** field in the declaration part and entering the new value. For a Boolean variable, change the value by single-clicking in the field.
- Click the inline monitoring field in the implementation part of the FBD/LD/IL editor and enter the new value.
- Click in the **prepared value** field in the Monitor window and enter the new value.



The command **Write values for all applications**, which affects all PLC projects in the TwinCAT project, is not included in a menu by default.

See also:

- [Command Force values \[► 960\]](#)
- PLC documentation: [Forcing and Writing Variables Values \[► 214\]](#)

17.8.17 Command Display Mode - Binary, Decimal, Hexadecimal

Function: The commands of the **Display** submenu are used to set the format for displaying the values during monitoring in online mode.

Call: **PLC** menu, context menu

Requirement: The PLC project is in offline or online mode.



The display formats "Binary" and "Hexadecimal" are unsigned, "Decimal" is signed.

See also:

- PLC documentation: [Monitoring Values \[► 222\]](#)
- PLC documentation: [Declaration Editor \[► 622\]](#)

17.8.18 Command Presentation of inheritance - Simple, Structured



Available from TC3.1 Build 4026

Function: The commands of the submenu **Presentation of inheritance** are used to set the format for presentation of the inheritance hierarchy of function blocks and structures when monitoring in online mode.

Call: PLC menu, context menu

Requirement: The PLC project is in offline or online mode.

Structured	The inheritance hierarchy of function blocks and structures is presented in a tree structure. The variables are presented as child nodes of the function block or structure in which they are declared.
Simple	The presentation is done as a flat list.

See also:

- PLC documentation: [Monitoring Values \[► 222\]](#)
- PLC documentation: [Declaration Editor \[► 622\]](#)

17.8.19 Command Create Localization Template

Function: The command opens the **Create Localization Template** dialog. Here you define which text information from the project is to be exported to a translation template of the file format pot.

Call: Menu PLC > Project Localization

Requirement: A project is open.

Create Localization Template dialog

The dialog is used to select the textual information to be included in the localization template.

Include the following information

Names	Texts such as dialog title, object names in the PLC project tree
Identifier	Variable identifier, example: "nCounter"
Strings	For example, "count" in the following declaration: sVar : STRING := 'count'
Comments	Comment texts in the programming blocks
Position information	<p>Select which positions of the text categories in the project selected above should be included in the translation file. The position information is always in the first line(s) of a section for a translation.</p> <p>Example:</p> <pre>#: D:\Proj1.project\Project_Settings:1 msgid "Projekteinstellungen" msgstr ""</pre> <ul style="list-style-type: none"> • "All": All found positions of the text are listed. • "First occurrence": The translation file includes the position in the project where the text to be translated occurs for the first time. • "None"
Create	The button opens the dialog for saving a file. The translation template is created in a text file of type POT Translation Template (*.pot). Each further create process generates a complete new template file.


17.8.20 Command Manage Localization

Function: The command opens the **Manage Localization** dialog. In the dialog, select the desired localization language or the original version of the project. You can also add or remove localization files *.<Language>.po to or from the project.

Call: Menu **PLC > Project Localization**

Requirement: A project is open.

Manage localization dialog

Available localizations	List of the localization files present in the project. Example: proj1-de.po proj1-en.po <Originalversion> The original version is always available. The project can only be edited in the original version.
Add	The button opens the dialog for selecting another po file from the file system.
Remove	The button removes the po file selected on the left from the project.
Standard localization	 The currently selected localization becomes the standard localization. The entry is displayed in bold.
Change localization	Use the button to switch to the currently selected localization.
OK	The project is displayed in the language of the country supplied by the file selected in Files. If you select <Original version> , the project will appear in the editable, unlocalized version.

17.8.21 Command Toggle Localization

Symbol: 

Function: The command switches between the currently set project localization and the <Original version>.

Call:

- Menu **Project > Localization**
- Button in the **Manage Localizations** dialog
- Button in the toolbar

Requirement: A project is open. A standard localization for the project is defined in the **Manage Localizations** dialog.

See also:

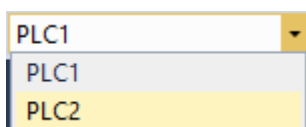
- [Command Manage Localization \[► 965\]](#)

17.8.22 Command Active PLC project

Function: Drop-down list for selecting the active PLC project. The currently focused project is automatically set as the active PLC project.

Call: TwinCAT PLC toolbar options

Requirement: The TwinCAT project contains several PLC projects.



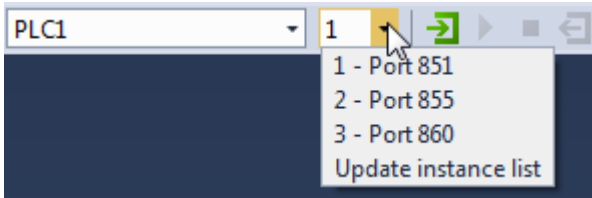
See also:

- [Command Active PLC instance \[► 966\]](#)
- [Command Login \[► 957\]](#)

17.8.23 Command Active PLC instance

Function: Drop-down list for selecting the active PLC instance of the corresponding PLC project.

Call: TwinCAT PLC toolbar options

**See also:**

- [Command Active PLC project \[► 965\]](#)
- [Command Login \[► 957\]](#)

17.9 Tools

17.9.1 Command Options

Function: The command opens the **Options** dialog for configuring TwinCAT options. These options define the behavior and appearance of the TwinCAT user interface. TwinCAT saves the current settings on the local system as default settings.

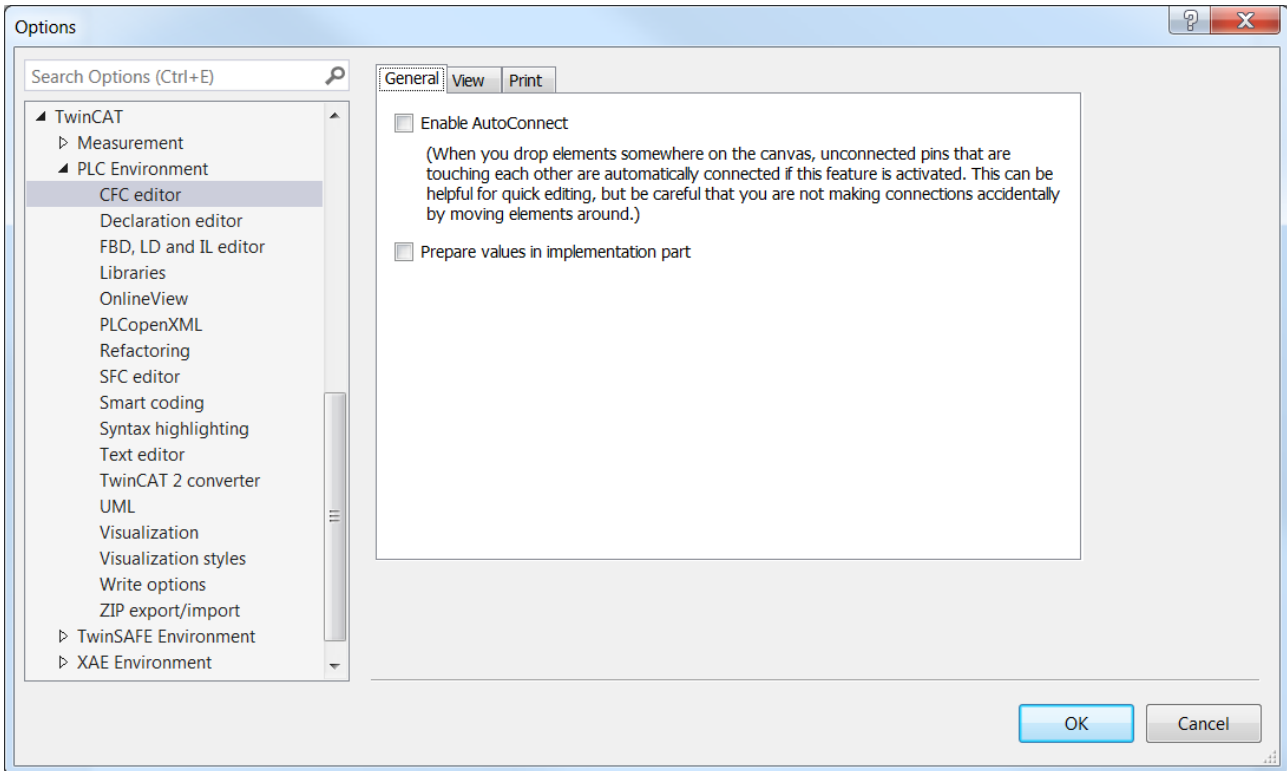
Call: Menu **Tools** > **Options** > **TwinCAT** > **PLC Environment**

17.9.1.1 Dialog Options - CFC Editor

Function: The dialog is used to configure the settings for editing and printing in the CFC editor.

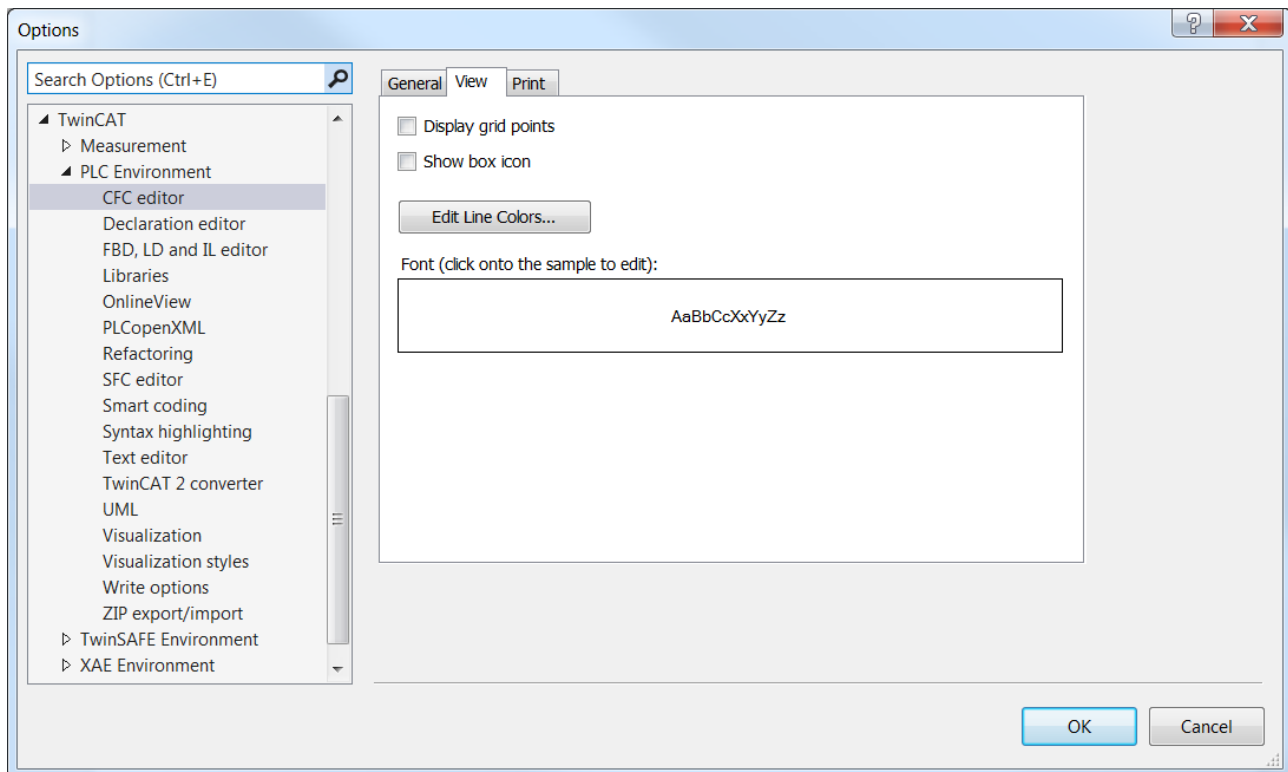
Call: **TwinCAT** > **PLC Environment** > **CFC editor**

General tab



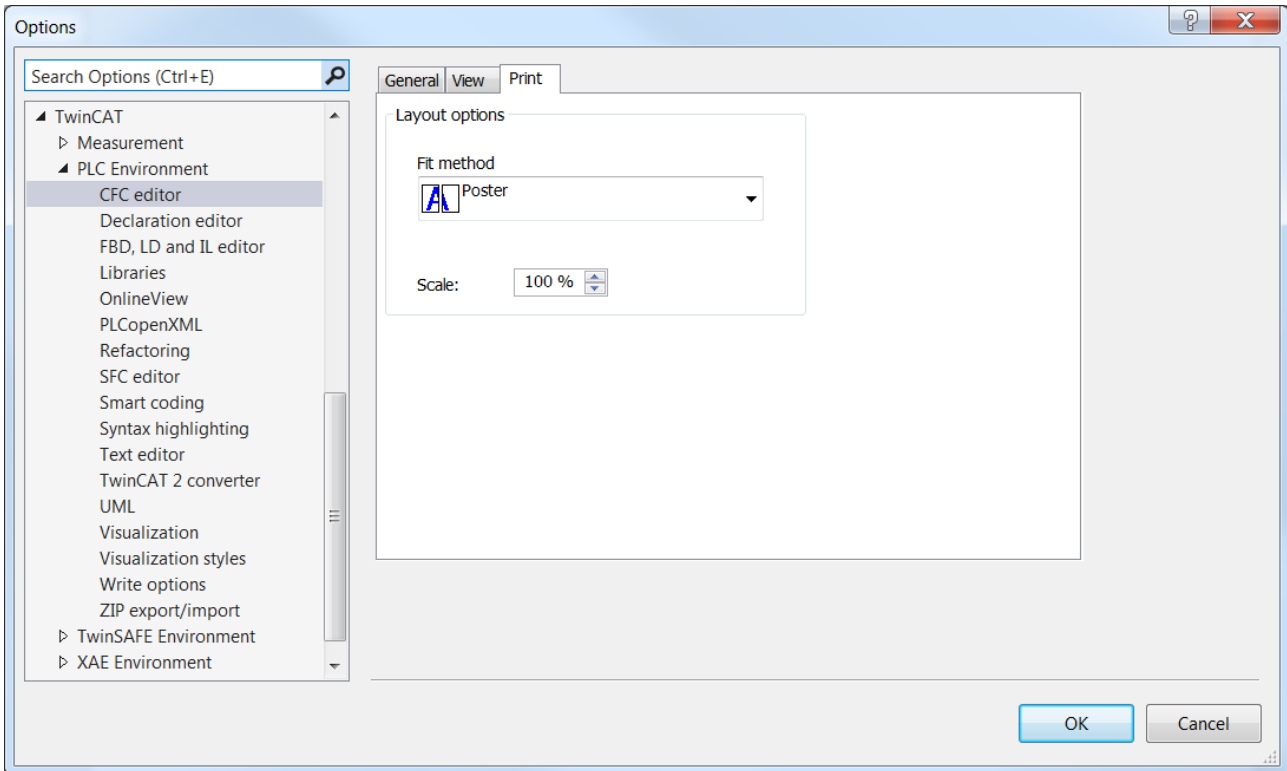
<p>Enable AutoConnect</p>	<p><input checked="" type="checkbox"/> : When you drag and paste a CFC element onto the editor's workspace, TwinCAT automatically connects unlinked pins that "touch" each other. Make sure that you do not create any unwanted connections when you move elements!</p>
<p>Prepare values in implementation part</p>	<p><input checked="" type="checkbox"/> : In online mode, you can also prepare variable values for writing and forcing in the implementation part of the CFC function block. In addition, TwinCAT displays the values that have just been prepared in the inline monitoring box of the variables in angle brackets.</p>

View tab



Display grid points	<input checked="" type="checkbox"/> : In the editor, grid points are validity areas at which you can position the elements.
Show box icon	<input checked="" type="checkbox"/> : TwinCAT displays function blocks in the CFC editor that are linked to a bitmap as symbols. Requirement: You have either created the link for a function block or a function in the object properties or loaded it using a library.
Edit Line Colors	Opens the Edit Line Colors dialog to define the colors for the connection lines depending on the current data type. The lines appear in these colors in offline and online mode, except TwinCAT overpaints these colors with bold black and blue lines to indicate Boolean data flow. <ul style="list-style-type: none"> • Add type: Adds a data type to the list. • Remove type
Font	Displays the font and button for changing the font.

Print tab



Layout options

Fit method	Page or poster
Scale	Possible values: 20% - 200%

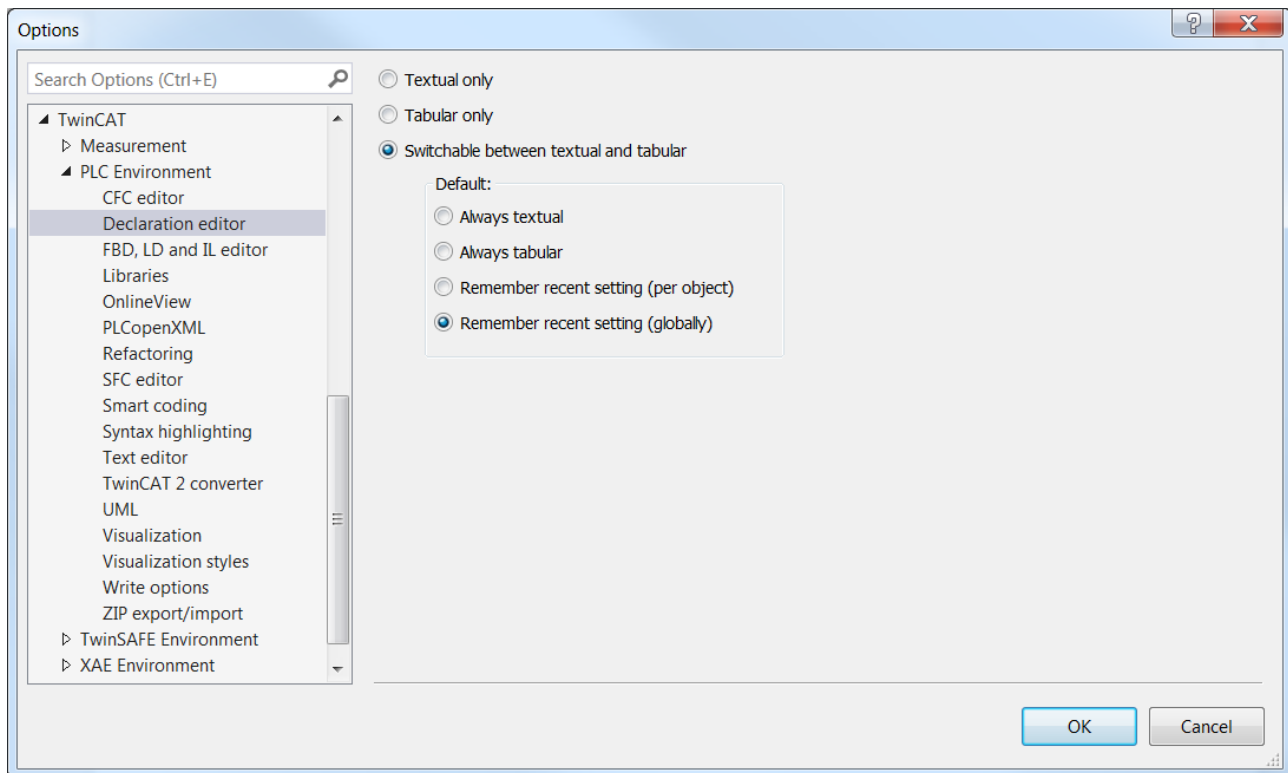
See also:



- PLC documentation: [Programming in CFC \[▶ 114\]](#)
- PLC documentation: [Programming languages and their editors \[▶ 622\]](#)

17.9.1.2 Dialog Options - Declaration Editor

Function: The dialog is used to configure the display settings for the declaration editor.

Call: TwinCAT > PLC Environment > Declaration editor



Textual only	Textual view of the declaration editor
Tabular only	Tabular view of the declaration editor
Switchable between textual and tabular	<p>The declaration editor provides two buttons to switch between the textual and tabular views:</p> <p> : Textual view</p> <p> : Tabular view</p> <p>The following option defines the view that appears by default when you open a programming object:</p> <ul style="list-style-type: none"> • Always textual • Always tabular • Remember recent setting (per object) • Remember recent setting (globally)

See also:

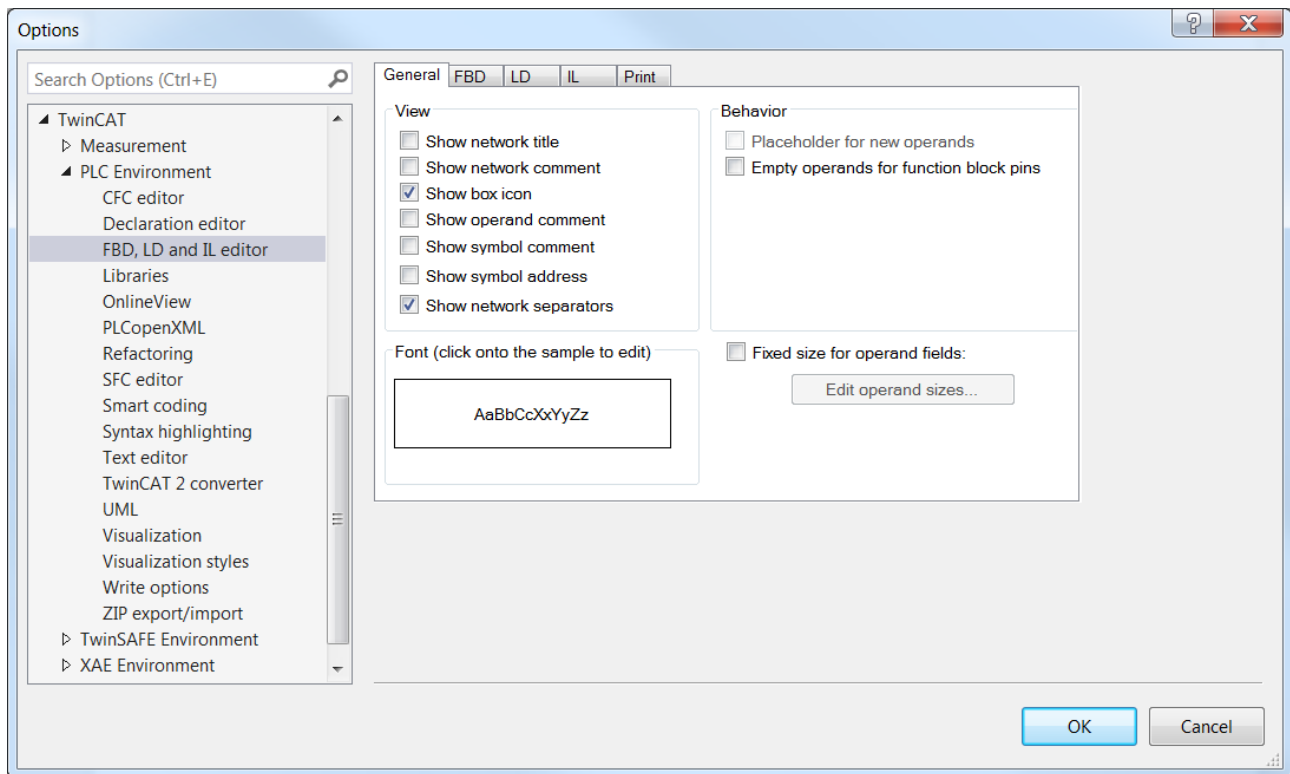
- PLC documentation: [Using the declaration editor \[► 70\]](#)

17.9.1.3 Dialog options - FBD, LD and IL

Function: The dialog is used to configure the presentation options for the FBD/LD/IL editor.

Call: TwinCAT > PLC Environment > FBD, LD and IL

General tab



View

Show network title	The network title is displayed in the upper left corner of the network.
Show network comment	The network comment is displayed in the upper left corner of the network. If TwinCAT also displays the network title, the comment appears in the line below.
Show box icon	The function block symbol is displayed in the block element in the FBD and LD editor. The standard operators also have symbols.
Show operand comment	TwinCAT displays the comment that you have assigned to a variable in the implementation part. The operand comment only refers to the local usage point of the variable, in contrast to the "symbol comment".
Show symbol comment	TwinCAT displays the comment that you have assigned to a variable or symbol in the declaration above the variable name. In addition to or instead of the symbol comment, you can also assign a local "operand comment".
Show symbol address	If an address is assigned to a symbol (variable), this address is displayed above the variable name.
Show network separators	A dividing line is displayed between the individual networks.

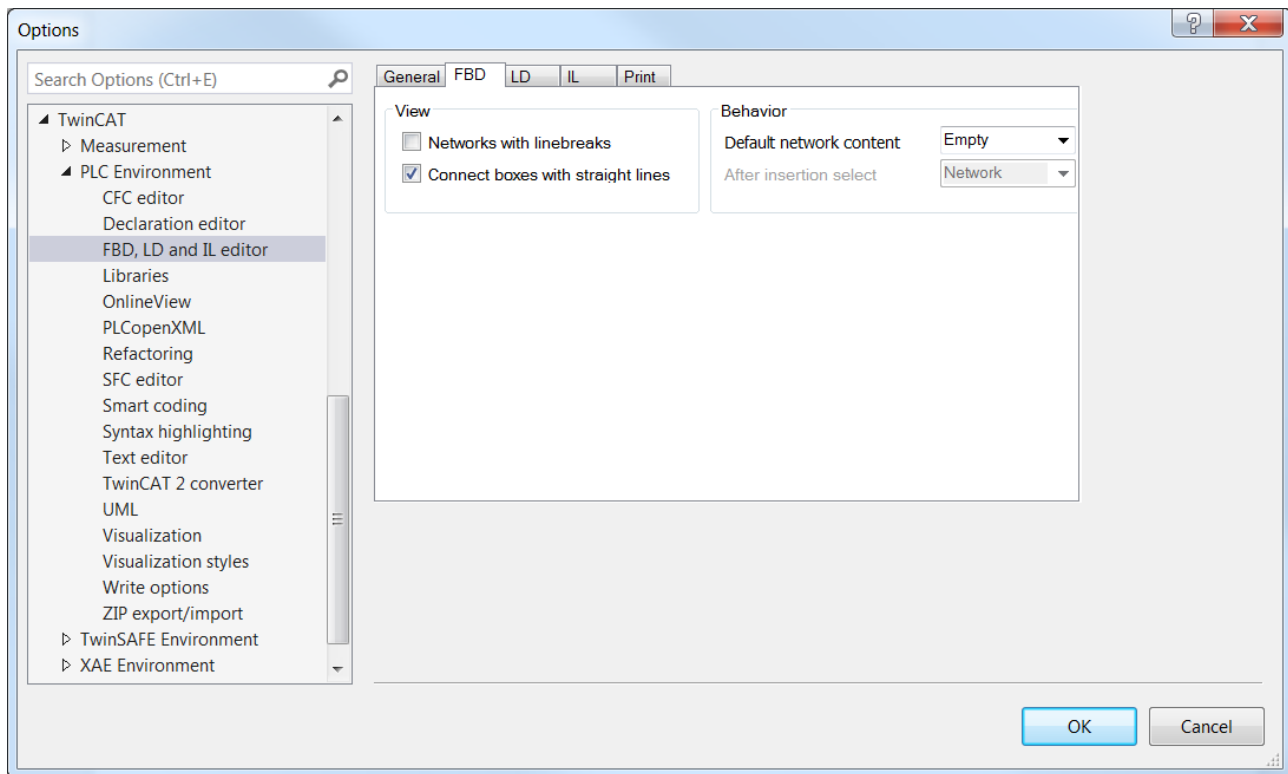
Behavior

Placeholder for new operands	The operand field of the pins of the new function blocks is left blank (instead of "???").
Empty operands for function block pins	Inserts empty operands instead of ???.

Font

Clicking on the input field opens the Font dialog.	
Fixed size for operand fields	<input checked="" type="checkbox"/> : Edit operand sizes can be enabled.
Edit operand sizes	The Operand Size dialog appears for setting the number of characters and lines.

FBD tab



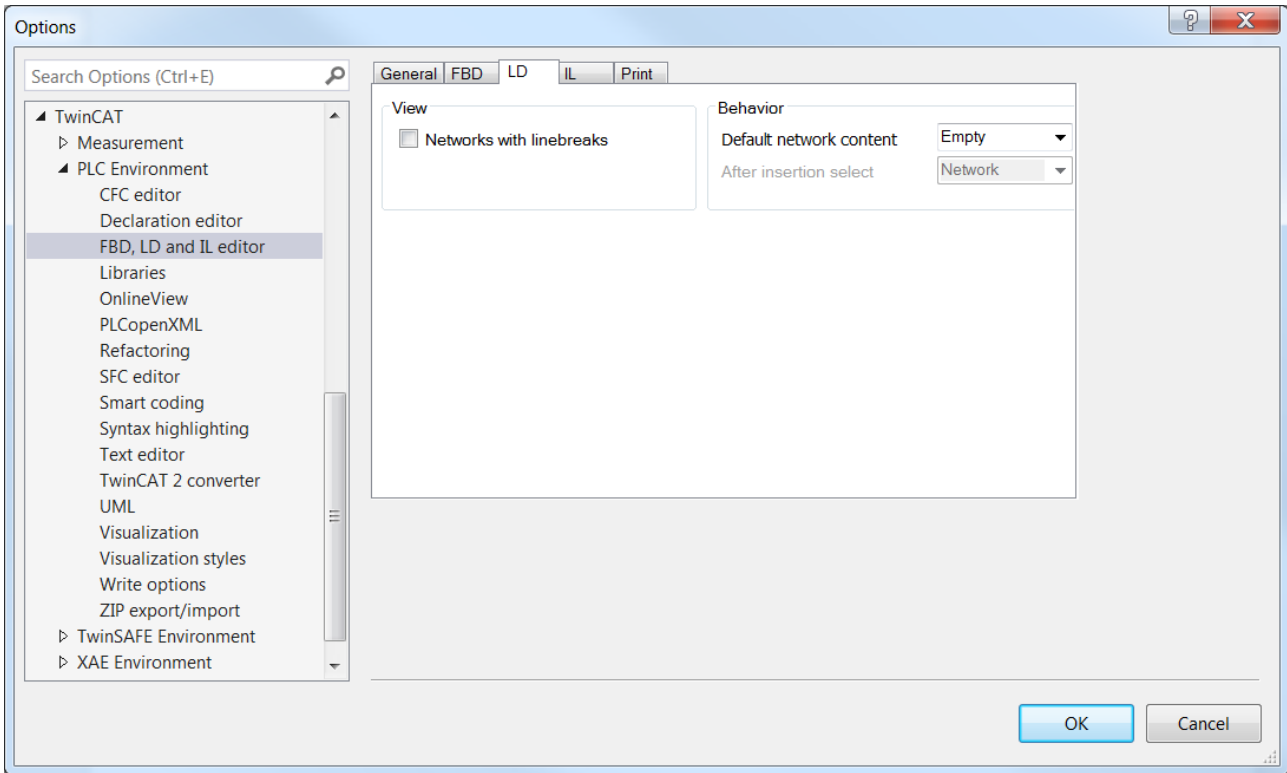
View

Networks with linebreaks	<input checked="" type="checkbox"/> : Representation of the networks with line breaks, so that TwinCAT can display as many function blocks as possible within the current width of the window.
Connect boxes with straight lines	<input checked="" type="checkbox"/> : Lines between the elements are given a fixed, short length.

Behavior

Default network content	Selection list: Contents of a new network.
After insertion select	Selection list: Element that TwinCAT selects after inserting a new network.

LD tab



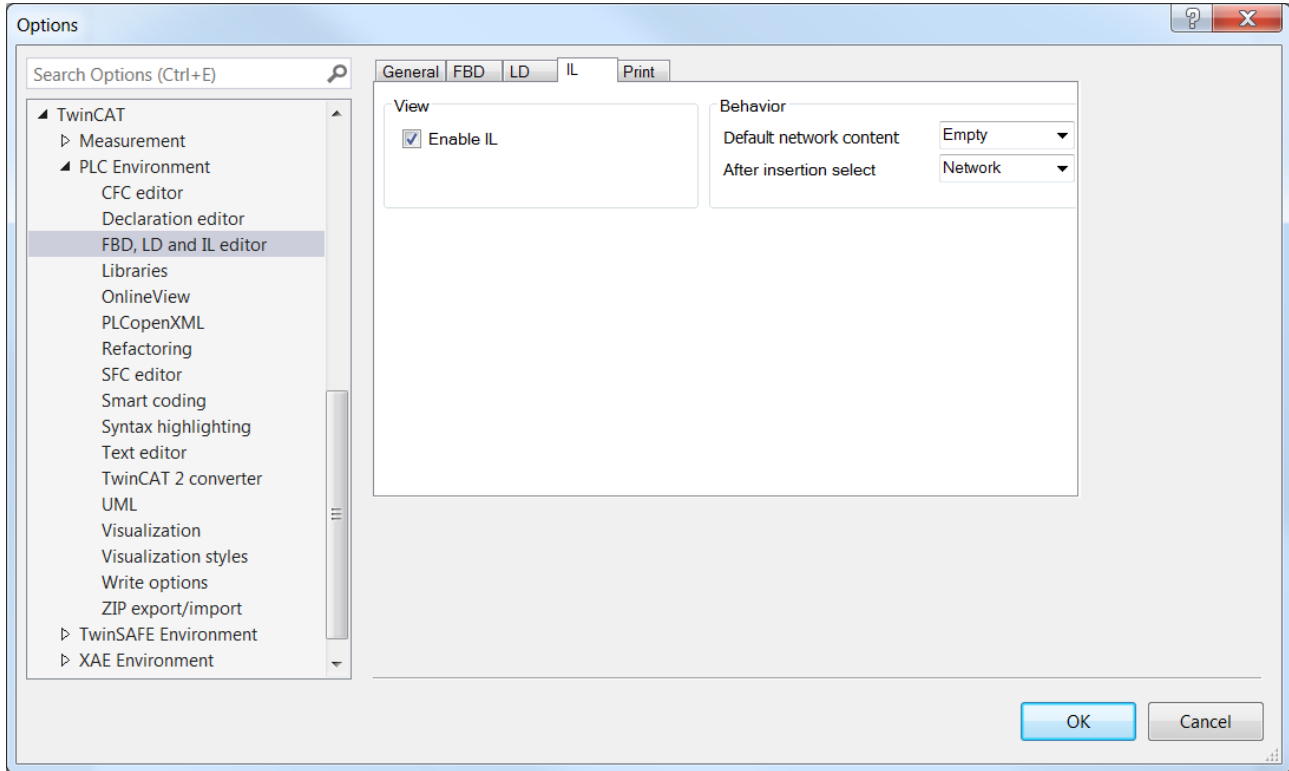
View

Networks with linebreaks	<input checked="" type="checkbox"/> : Representation of the networks with line breaks, so that TwinCAT can display as many function blocks as possible within the current width of the window.
--------------------------	--

Behavior

Default network content	Selection list: Contents of a new network.
After insertion select	Selection list: Element that TwinCAT selects after inserting a new network.

IL tab



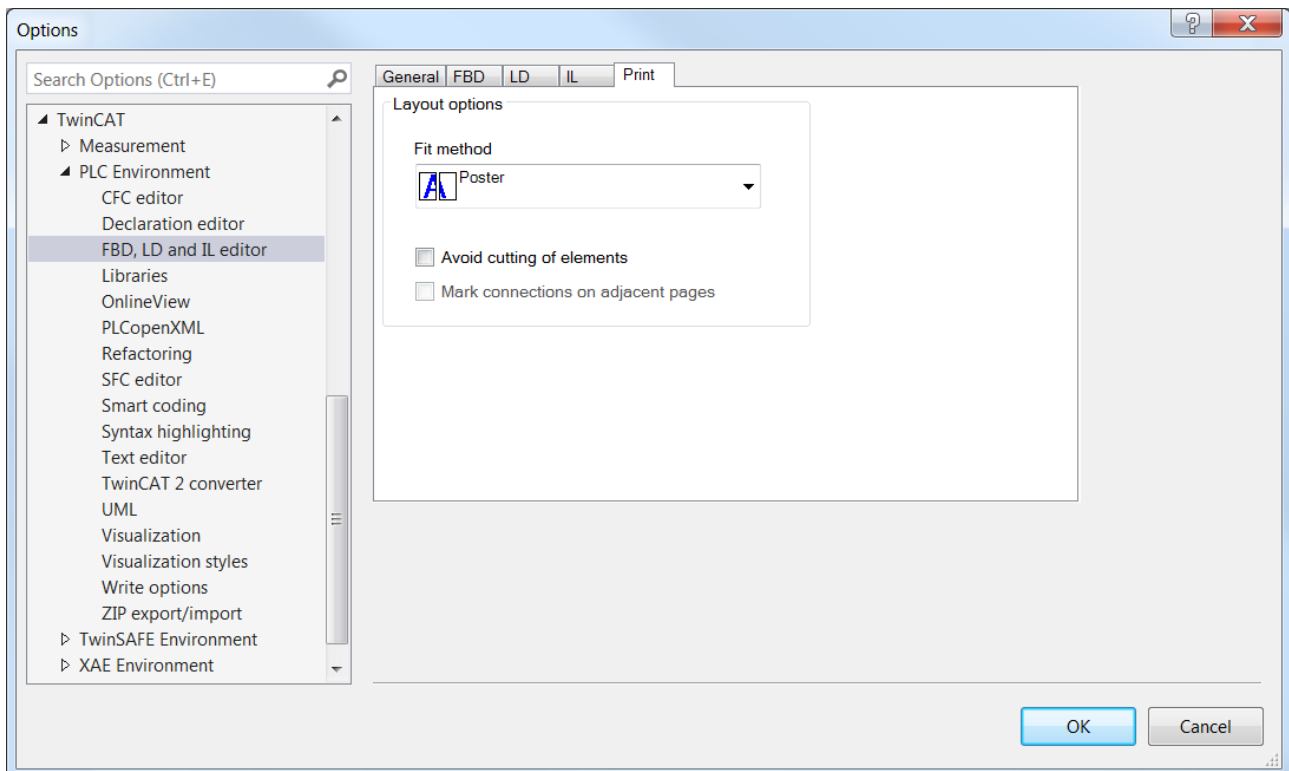
View

Enable IL	The implementation language IL is available in the development system.
-----------	--

Behavior

Default network content	Selection list: Contents of a new network.
After insertion select	Selection list: Element that TwinCAT selects after inserting a new network.

Print tab



Layout options

Fit method	Selection list for size adjustment.
Avoid cutting of elements	Items that do not fit on the page are printed on the next page.
Mark connections on adjacent pages	Can be enabled if Avoid cutting of elements is enabled.

See also:

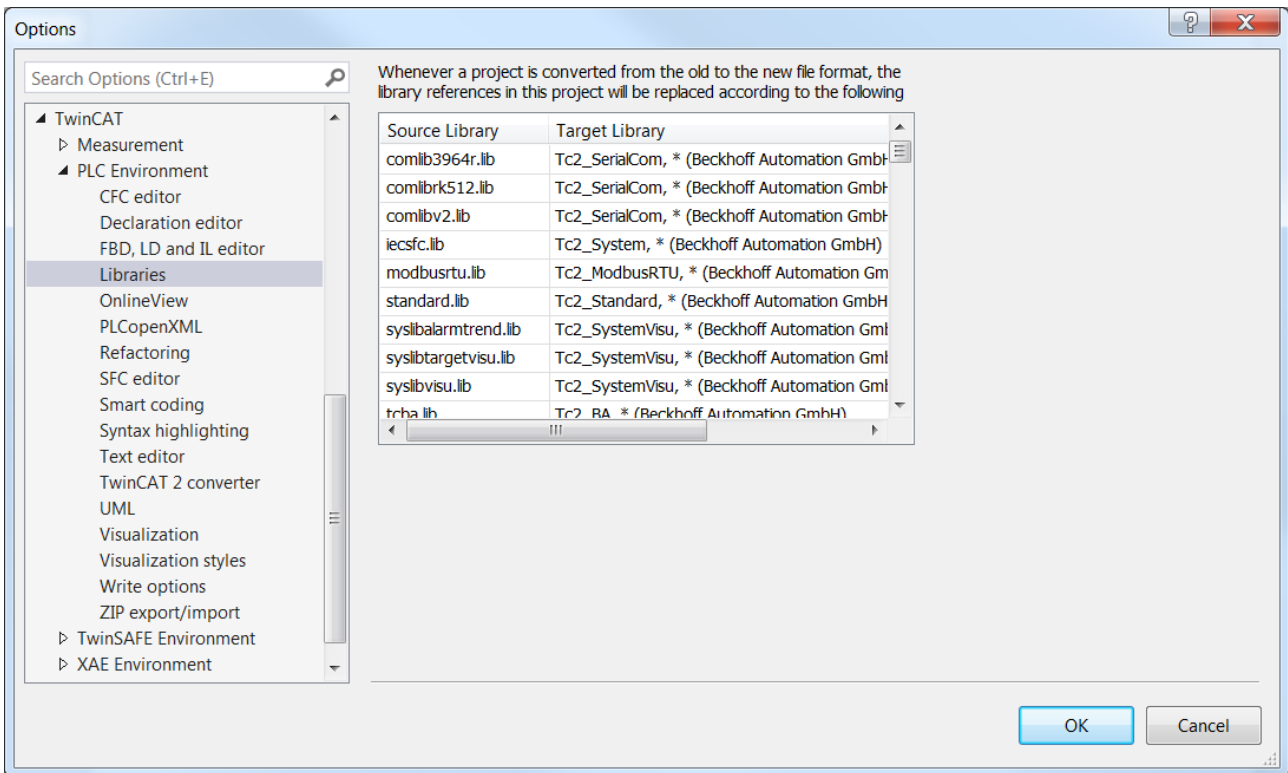
- PLC documentation: [FBD/LD/IL](#) [▶ 108]
- PLC documentation: [Programming languages and their editors](#) [▶ 622]

17.9.1.4 Dialog Options - Libraries

Function: The dialog contains settings for libraries.

Call: TwinCAT > PLC programming environment > Libraries

Conversion tab



The tab is used to manage the mappings of library references that TwinCAT uses when converting an old project. If you have not yet saved a mapping for a particular library, you must redefine the mapping each time you open an old project that includes this library.

A mapping defines how a library reference looks like after conversion of the project to the current format. There are three options:

- You retain the reference. This means that TwinCAT also converts the library into the current format (*.library) and installs it in the local library repository.
- You replace a reference with another reference. This means that one of the installed libraries replaces the previously referenced library.
- You delete the reference. This means that the converted project no longer integrates the library.

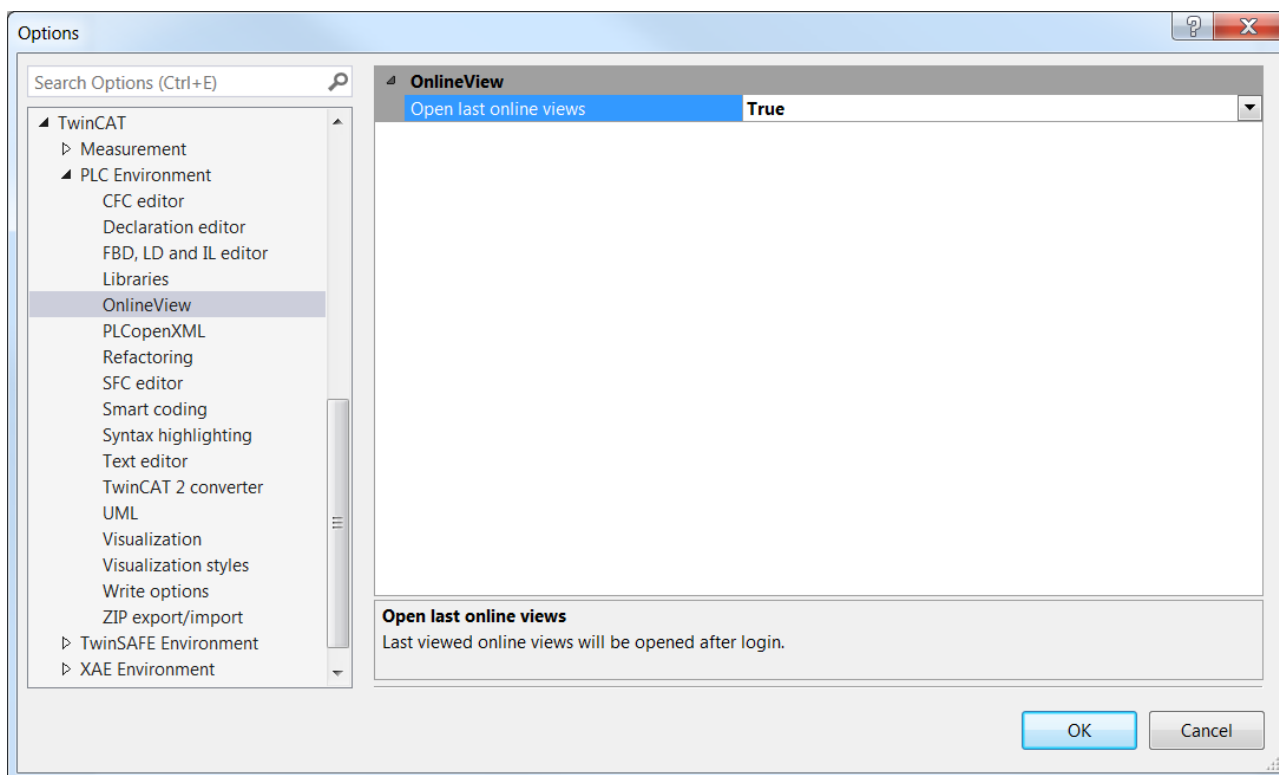
During the next conversion of an old project, TwinCAT applies all listed mappings to its library references. This means that you have to repeat the mapping definition if the same library is included again in a project to be converted. You can enter a new mapping in the last line.

Source Library	Path of the library that is included in the project before the conversion. Double-clicking an entry makes the field editable and the Input Assistant button appears.
Target Library	Name and location of the library to be included in the project after conversion. Double-clicking on an entry opens the "Set target system library" dialog.

See also:

- PLC documentation: [Using libraries](#) [▶ 266]

17.9.1.5 Dialog Options - Online View



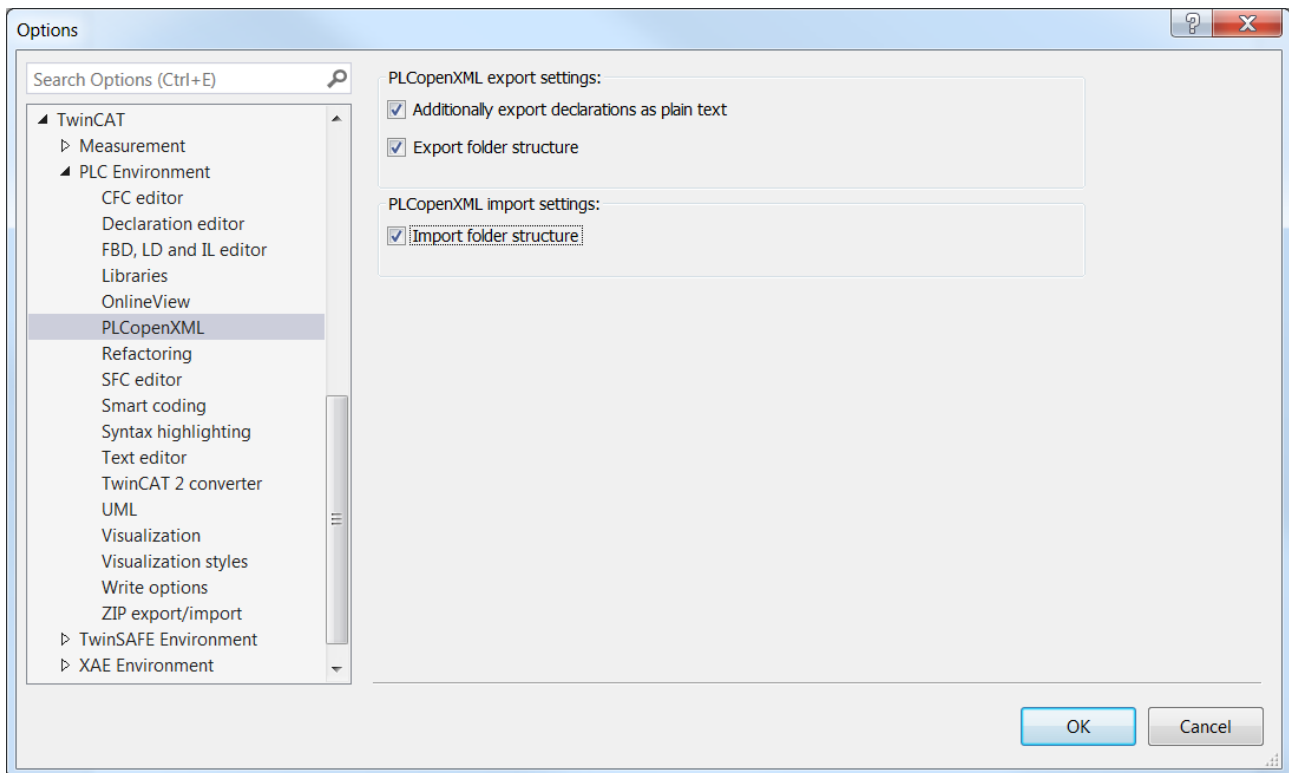
OnlineView

Open last online views	<ul style="list-style-type: none"> • TRUE (default setting): On login the editor windows of the previous online session open. The current offline view remains open. • FALSE: On login the offline view remains open. The editor windows of the previous online session are discarded and not reopened.
------------------------	---

17.9.1.6 Dialog Options - PLCopenXML

Function: The dialog contains settings for the behavior of TwinCAT during PLCopenXML export or import.

Call: TwinCAT > PLC Environment > PLCopenXML



PLCopenXML export settings

<p>Additionally export declarations as plain text</p>	<p>By default, TwinCAT splits the declaration parts into individual variables in accordance with the PLCopenXML schema, resulting in loss of formatting and some commentary information.</p> <p><input checked="" type="checkbox"/> : Formatting and comments are retained. TwinCAT also writes the plain text of the exported declaration part to the PLCopenXML file, thus extending the PLCopenXML schema.</p>
<p>Export folder structure</p>	<p><input checked="" type="checkbox"/> : TwinCAT also exports the folders if they contain one of the selected objects. This is a TwinCAT-specific extension to the PLCopenXML schema.</p>

PLCopenXML import settings

<p>Import folder structure</p>	<p><input checked="" type="checkbox"/> : If the import file contains information about the folder structure of the objects, TwinCAT imports this structure.</p> <p><input type="checkbox"/> : TwinCAT imports objects without a structure.</p>
--------------------------------	--

See also:

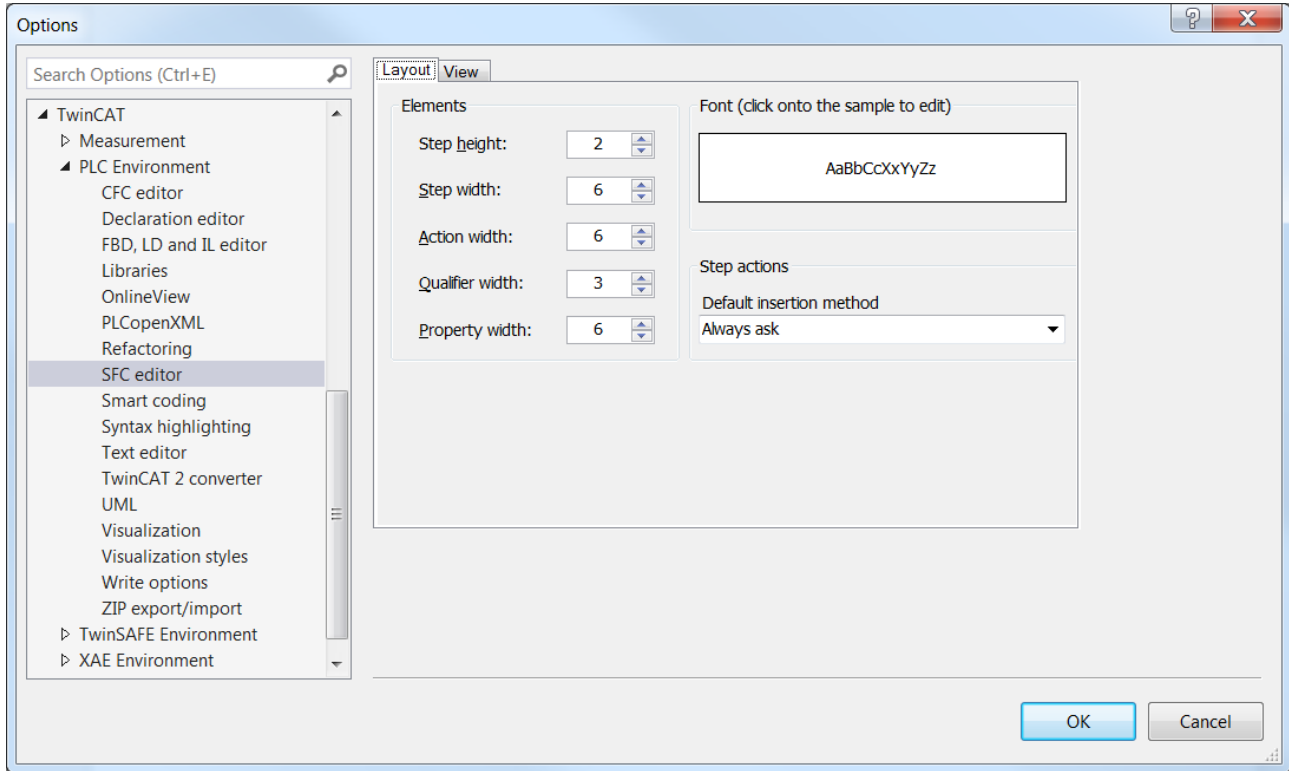
- Command Export PLCopenXML
- Command Import PLCopenXML
- PLC documentation: [Exporting and importing a PLC project \[► 62\]](#)

17.9.1.7 Dialog Options - SFC editor

Function: The dialog is used to configure the settings for the SFC editor.

Call: TwinCAT > PLC Environment > SFC editor

Layout tab



Elements

Defines the sizes of the SFC elements Step, Action, Qualifier and Property. Specify the values in grid units. 1 grid unit = font size currently set in the text editor options (Text area / Font). The settings always take effect immediately in all currently open SFC editor windows.

Step height	Possible values: 1-100
Step width	Possible values: 2-100
Action width	Possible values: 2-100
Qualifier width	Possible values: 2-100
Property width	Possible values: 2-100

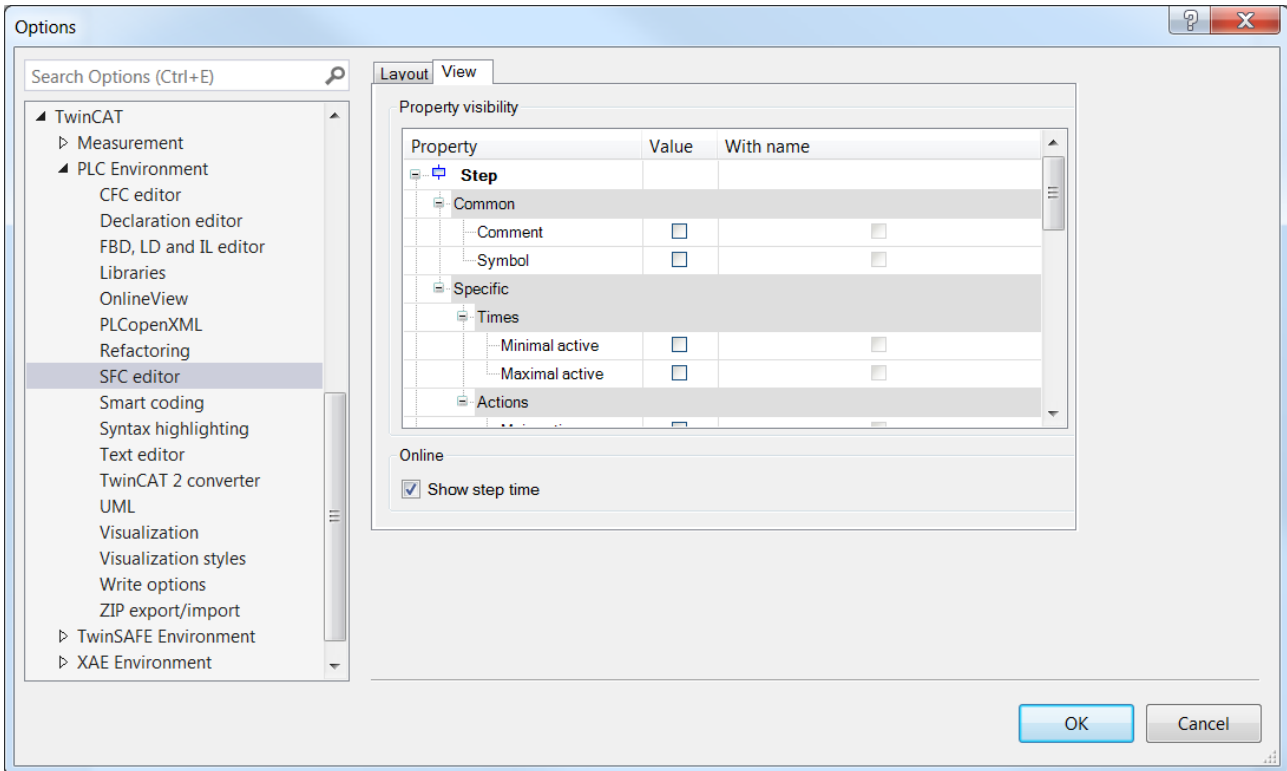
Font

The sample text shows the currently set font. Click on it to change the font.

Step actions

Default insertion method	<ul style="list-style-type: none"> • Always ask • Duplicate implementation • Copy reference
--------------------------	--

View tab



Property visibility

Lists the item properties of the Common and Specific categories and defines display options.	
Property	Defines the item properties that are shown next to the item in the SFC diagram.
Value	<input checked="" type="checkbox"/> : Display of the property value.
With name	<input checked="" type="checkbox"/> : Display of the property value with name.

Online

Show step time	<input checked="" type="checkbox"/> : TwinCAT displays the step time in online mode to the right of the steps.
----------------	--

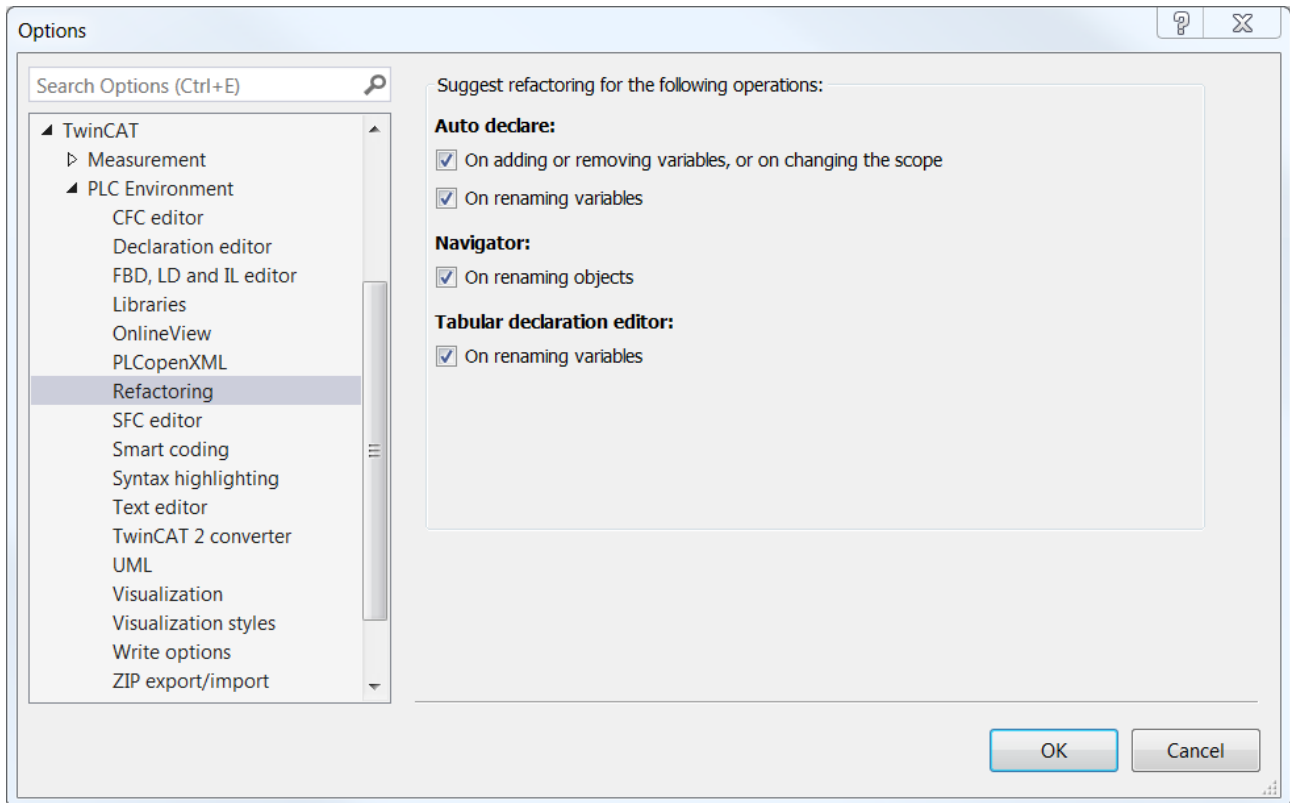
See also:

- PLC documentation: [Sequential Function Chart \(SFC\) \[► 124\]](#)
- PLC documentation: [Programming languages and their editors \[► 622\]](#)

17.9.1.8 Dialog Options - Refactoring

Function: The dialog is used to define the operations in the project for which refactoring is automatically proposed. The refactoring functionality supports your enhancement efforts.

Call: TwinCAT > PLC programming environment > Refactoring



Auto declare

If you change the name of a variable or add an input or output variable by calling the autodeclaration (**Auto Declare** dialog) , the option **Apply changes using refactoring**, is automatically enabled. After confirming the dialog, the **Refactoring** dialog opens and you can change the variable project-wide.

When adding or removing variables, or for changing the scope

: Add a new input or output variable via the autodeclaration (**Auto Declare** dialog) or delete the name of a variable in the autodeclaration and close the dialog with **OK**. This will open the **Refactoring** dialog to add or remove the variable project-wide

On renaming variables

: Rename the name in the auto declaration (**Auto Declare** dialog) and close the dialog with **OK**. This opens the **Refactoring** dialog to rename the variable project-wide.

Navigator

On renaming objects

: If you change the name of an object in the PLC project tree, a prompt appears asking whether TwinCAT should carry out "Automatic refactoring".

Tabular declaration editor

On renaming variables

: If you change the name of a variable in the tabular declaration editor, a prompt appears asking whether TwinCAT should carry out "Automatic refactoring" for the renaming.

UML class diagram

Options for refactoring support for changes made in the class diagram editor.

When adding or removing variables

: When you add or remove variables in the VAR_INPUT, VAR_OUTPUT, and VAR_IN_OUT sections in the class diagram, refactoring is supported.

When renaming a function block

: If you change a function block name in the class diagram, refactoring is supported.

When renaming variables or properties

: If you rename a variable or a property in the class diagram, refactoring is supported.

If the **Skip Refactoring Preview** option is enabled in the UML Options, refactoring may be performed on all affected locations in the project without first being displayed in the **Refactoring** dialog, depending on the case. (see [Dialog Options - UML](#) [▶ 990])

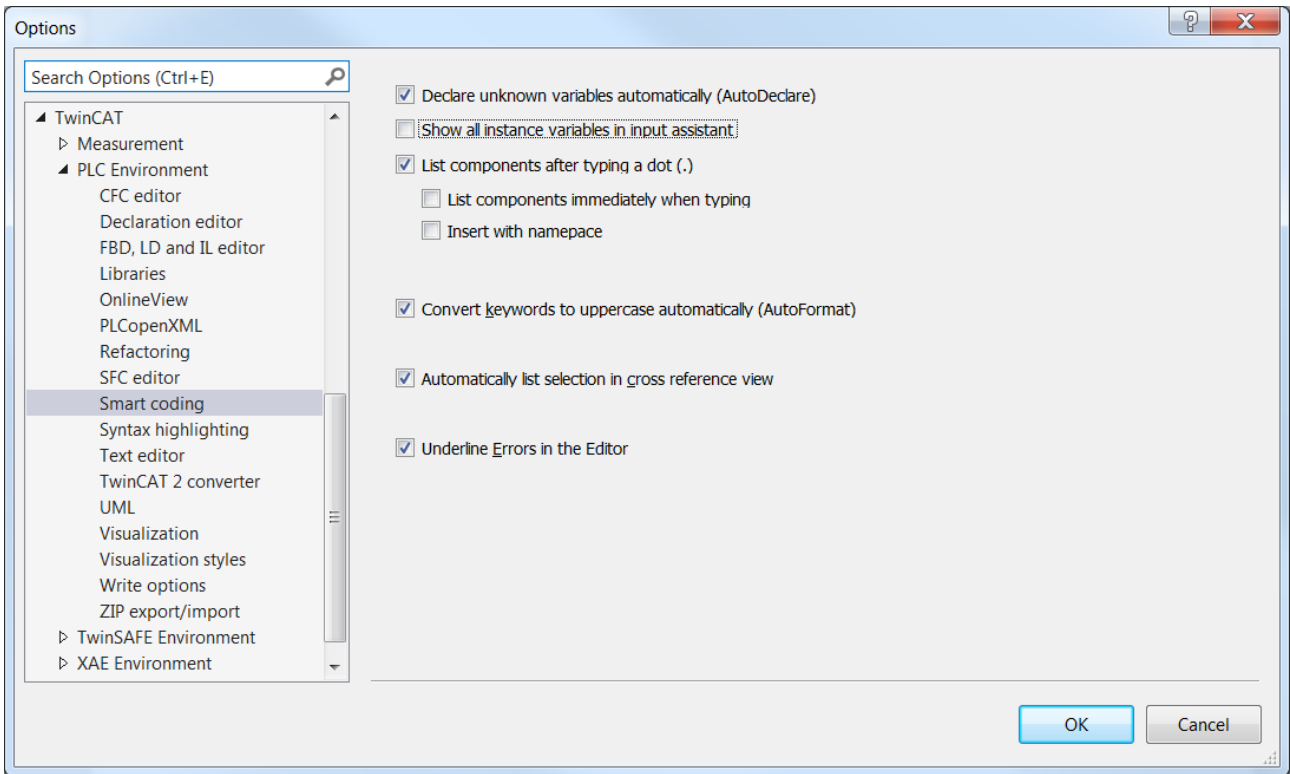
See also:

- [Command Auto Declare](#) [▶ 875]
- [Command Rename '<variable>'](#) [▶ 886]
- [Command Add '<variable>'](#) [▶ 887]
- [Command Remove '<variable>'](#) [▶ 889]
- PLC documentation: [Refactoring](#) [▶ 160]

17.9.1.9 Dialog Options - Smart coding

Function: The dialog is used to configure the settings that make it easier to enter code.

Call: TwinCAT > PLC programming environment > Smart Coding



Declare unknown variables automatically (AutoDeclare)	<input checked="" type="checkbox"/> : The Auto Declare dialog opens as soon as you have entered an undeclared identifier in a programming language editor and exited the input line. For the AutoDeclare function to be available in the ST editor as well, the option Enable for the ST editor must also be enabled as of build 4026.
Enable for the ST editor	Prerequisite: The option Declare unknown variables automatically (AutoDeclare) is enabled. <input checked="" type="checkbox"/> : The AutoDeclare function is also available in the ST editor. <input type="checkbox"/> : The AutoDeclare function is not available in the ST editor. (Available from Build 4026)
Show all instance variables in input assistant	<input checked="" type="checkbox"/> : The List components function also offers the local variables of a function block instance for selection. <input type="checkbox"/> : The List components function only offers the input and output variables of an FB instance for selection.
List components after typing a dot (.)	<input checked="" type="checkbox"/> : Enables the List components function. This means: If you enter a dot at a point where TwinCAT expects an identifier, a selection list with input options appears.
List components immediately when typing	Requirement: Option List components after typing a dot (.) is enabled. <input checked="" type="checkbox"/> : After entering any character string, a selection list of the available identifiers and operators appears
Insert with namespace	<input checked="" type="checkbox"/> : TwinCAT automatically inserts the namespace before the identifier.
Convert keywords to uppercase automatically (AutoFormat)	<input checked="" type="checkbox"/> :TwinCAT automatically writes all keywords in capital letters.
Automatically list selection in cross reference view	<input checked="" type="checkbox"/> : The Cross Reference List automatically displays the references of the variables/POUs/DUTs that are currently selected or in which the cursor is positioned.
Underline errors in the editor	<input checked="" type="checkbox"/> : Erroneous or unknown program code is underlined. (Available from Build 4026)
Highlight symbols	<input checked="" type="checkbox"/> : All the places of use of a symbol on which the cursor is positioned are highlighted in color within the editor. This allows you to quickly identify cross-references within the editor. (Available from Build 4026)

See also:

- PLC documentation: [Programming languages and their editors \[▶ 622\]](#)
- PLC documentation: [Using the input assistant \[▶ 135\]](#)
- PLC documentation: [Find locations where the cross reference list is used \[▶ 157\]](#)

17.9.1.10 Dialog: Options – Ladder editor

Symbol: 

Function: The dialog is used to configure the presentation options for the Ladder Diagram editor.

Call: TwinCAT > PLC Environment > Ladder editor

General tab**View**

Show network title	The network title is displayed in the upper left corner of the network.
Show network comment	The network comment is displayed in the upper left corner of the network. If TwinCAT also displays the network title, the comment appears in the line below.
Show box icon	The box icon is displayed in the box element in the Ladder editor. The standard operators also have symbols.
Show operand comment	TwinCAT displays the comment that you have assigned to a variable in the implementation part. The operand comment only refers to the local usage point of the variable, in contrast to the "symbol comment". The comment is automatically truncated, depending on available space.
Show symbol comment	TwinCAT displays the comment that you have assigned to a variable or symbol in the declaration above the variable name. In addition to or instead of the symbol comment, you can also assign a local "operand comment".
Show symbol address	If an address is assigned to a symbol (variable), this address is displayed above the variable name.

Operand size

Maximum number of displayed lines	Maximum number of lines of the operand name which are displayed.
Maximum average characters per line	Maximum number of characters per line for displaying the operand name

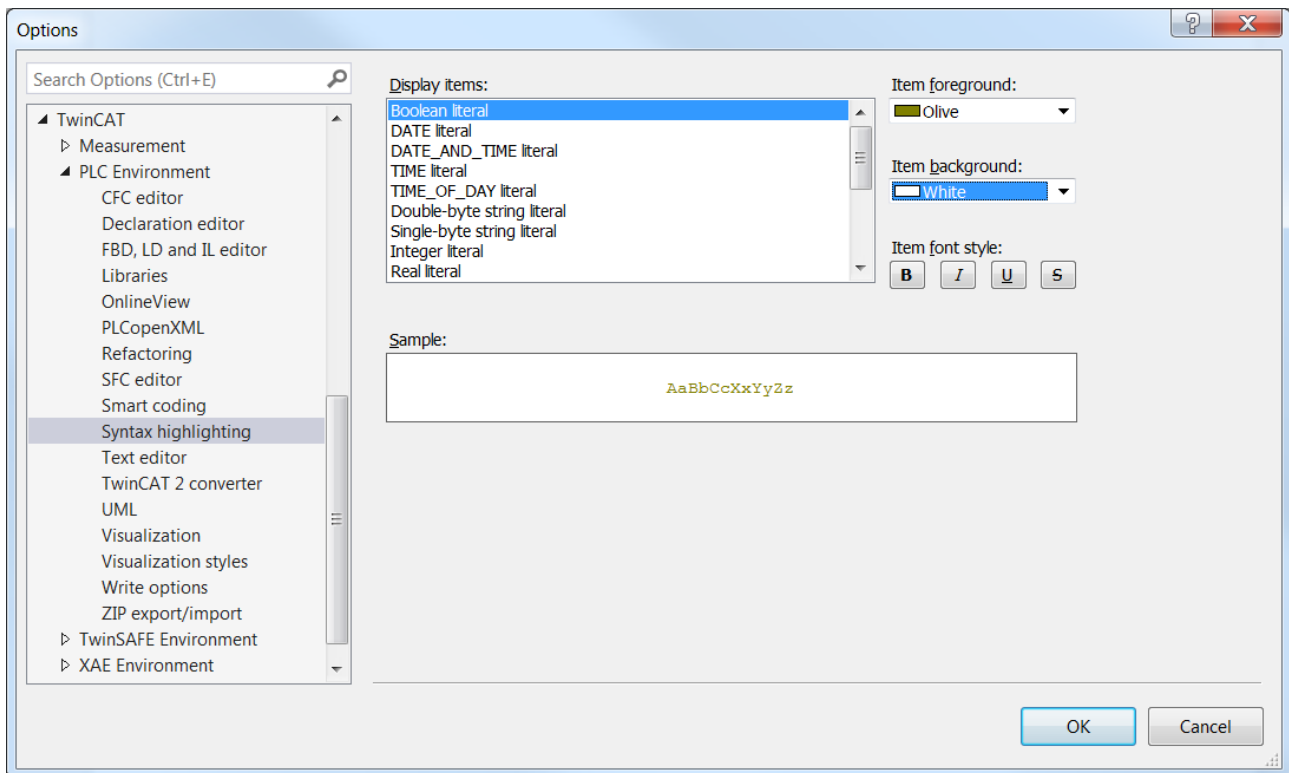
Monitoring tab

Displayed digits for floating-point numbers	Selection list for the number of digits Example: The value <i>750.15</i> for 5 set digits becomes <i>7.5e+02</i> for 2 set digits.
Displayed string length	Selection list for the number of characters

17.9.1.11 Dialog Options - Syntax highlighting

Function: The dialog is used to configure the color and font settings for the text elements of an editor (e. g. operands, pragmas).

Call: TwinCAT > PLC Environment > Syntax highlighting



Display items	Selection list for text items
Item foreground	Foreground color of the text item
Item background	Background color of the text item
Item font style	Text item font style (bold, italic, underlined, strikethrough)
Sample	The sample text shows the current settings in a preview

17.9.1.12 Dialog Options - Text editor

Function: The dialog contains settings for displaying and working in a text editor.

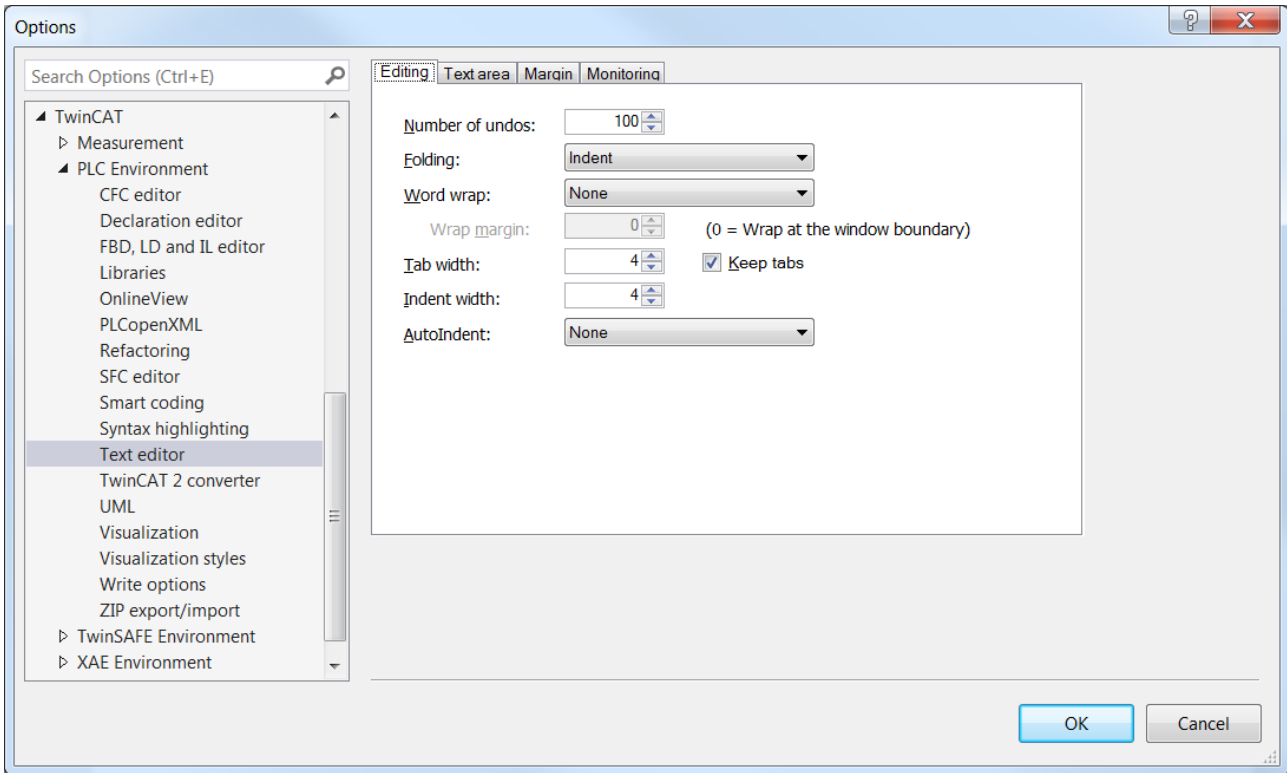
Call: TwinCAT > PLC programming environment > Text editor

Theme tab

In this tab you set the desired theme for the ST Editor interface design. (Available from Build 4026)

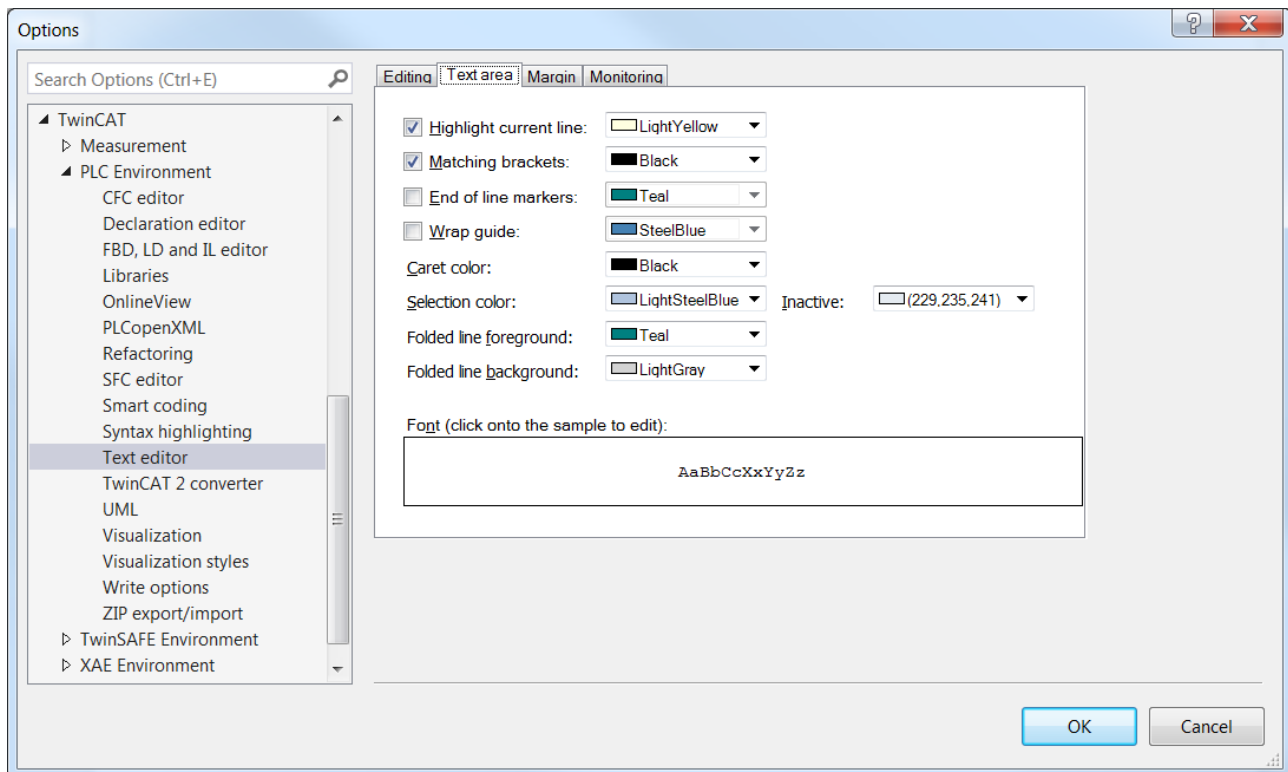
Theme	Color scheme for the text editor. The selected theme is displayed in the Preview window.
-------	--

Editing tab



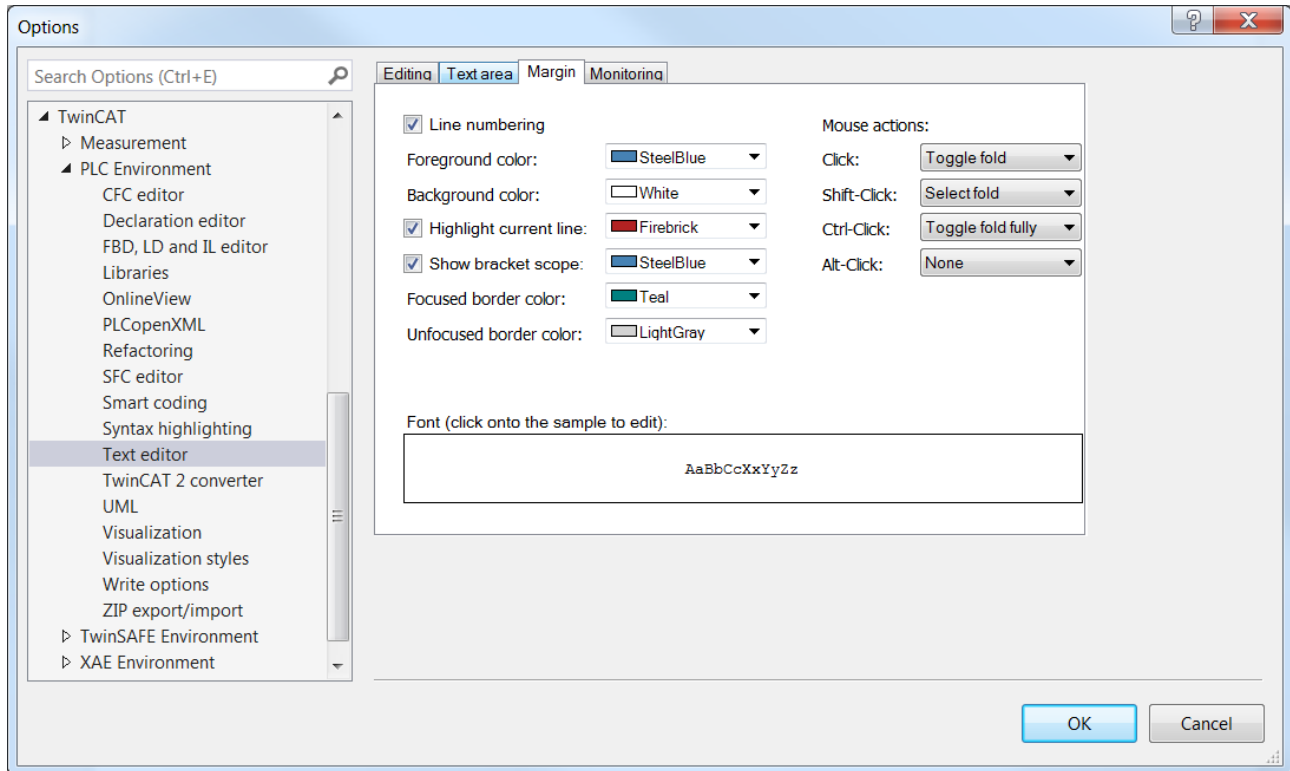
Number of undos	Maximum number of steps for which the command Edit > Undo is available.
Folding	<p>Defines the structuring of the code by indentations.</p> <p>When you select an indentation, you can expand or collapse the indentation section using a plus or minus sign to the left of the first line of each section.</p> <ul style="list-style-type: none"> • Indentation: TwinCAT combines all lines that are indented relative to the previous line in one indentation unit. • Explicit: You explicitly use comments to identify the code section to be grouped in an indentation unit: The section must be preceded by a comment containing 3 opening curly brackets "{{{", and followed by a comment containing 3 closing curly brackets "}}}". The comments can contain additional text. Sample: <pre data-bbox="550 629 1101 1003"> 1 IF nVar1=1 2 ///comment {{{ 3 THEN 4 nVar2:=2; 5 ELSE nVar2:=10; 6 END_IF 7 ///}}} 8 nVar1:=nVar1+1; 1 IF nVar1=1 2 ///comment {{{ [5 lines] 3 4 5 6 7 8 nVar1:=nVar1+1; </pre>
Word wrap	<ul style="list-style-type: none"> • Soft: The line break occurs at the edge of the editor window if 0 is entered for the wrap margin. • Hard: The line break occurs after the number of characters specified at the wrap margin.
Tab width	Number of characters
Keep tabs	<input checked="" type="checkbox"/> : The space that you have inserted with the [Tab] key will not subsequently be converted by TwinCAT into individual spaces.
Indent width	If you have activated Smart or Smart with code completion for the Auto Indent option, TwinCAT inserts the number of spaces at the beginning of the line.
Auto Indent	<ul style="list-style-type: none"> • Do not indent automatically • Block: A new line automatically adopts the indentation of the previous line. • Intelligent: Lines that follow a line containing a keyword (for example, VAR) automatically indent by the specified indent width. • Smart with code completion: Indentation as with the Smart option, in addition TwinCAT inserts the final keyword (e.g. END_VAR).

Text area tab



Highlight current line	<input checked="" type="checkbox"/> : The line in which the cursor is positioned is highlighted.
Matching brackets	<input checked="" type="checkbox"/> : If the cursor is positioned before or after a bracket within a code line, TwinCAT marks the corresponding closing or opening bracket with a frame.
Line markers	<input checked="" type="checkbox"/> : TwinCAT marks the end of each editor line with a small horizontal line after the last character of the line (including spaces).
Wrap guide:	<input checked="" type="checkbox"/> : If a soft or hard line break is enabled, the defined wrap guide is displayed by a vertical line.
Caret color	Color of the cursor character
Selection color	Color of the selected text area
Inactive	Color of a selection if the corresponding window is not active (focus is on another window).
Folded line foreground	Color of the header line of a closed indented section in the code
Folded line background	Header line of a closed indented section in the code is highlighted in color.
Font	A click on the field opens the standard dialog for configuring the font.

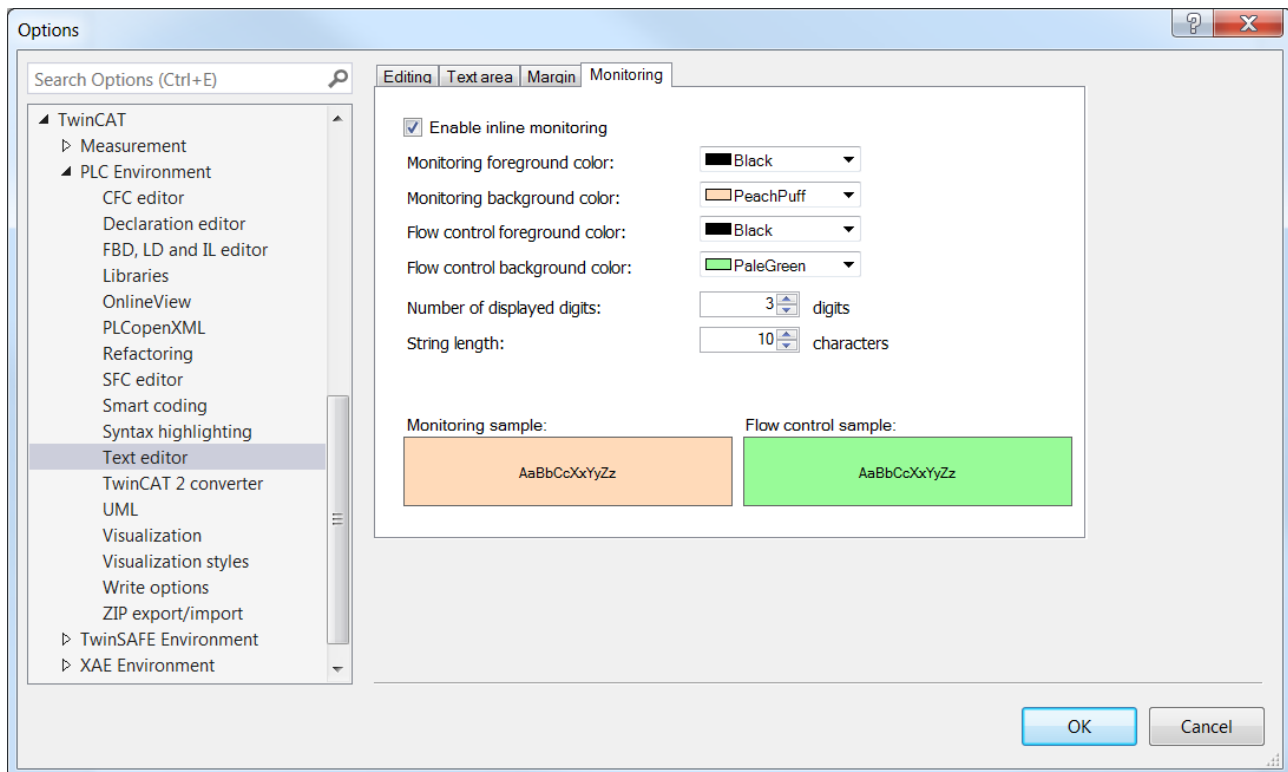
Margin tab



Settings for the left margin of the text editor window, which is separated by a vertical line from the input area:

Line numbering	<input checked="" type="checkbox"/> : Display of the line numbers in the declaration and implementation part, each starting with 1
Foreground color	Line number color
Background color	Margin color
Highlight current line	<input checked="" type="checkbox"/> : The line number of the line where the cursor is located is highlighted.
Show bracket scope	<input checked="" type="checkbox"/> : Bracketing comprises the lines between the keywords that open and close a construct, such as IF and END_IF. If the option is enabled and the cursor is positioned before, after or in one of the keywords of a construct, the bracketing area is indicated by a square bracket in the margin.
Focused border color	Color of the dividing line between the margin and the input area
Unfocused border color	Color of the dividing line between the margin and input area of the currently inactive part of the window
Mouse actions	One of the following actions can be assigned to each of the specified mouse actions or mouse button shortcuts. TwinCAT executes the actions when you execute the mouse action on the plus or minus sign before the header line of a bracketed area: <ul style="list-style-type: none"> • None: The mouse action does not trigger any action. • Select fold: TwinCAT selects all the lines of the bracketed area. • Toggle fold: TwinCAT opens or closes the bracketed area, or, in case of nested bracketing, the first level of the bracketed area. • Toggle fold fully: TwinCAT opens or closes all levels of a nested bracketed area.

Monitoring tab



Settings for displaying the monitoring fields

Enable Inline Monitoring	<input checked="" type="checkbox"/> : The monitoring fields are displayed after the variables in online mode
Monitoring foreground color	Representation of the value in the monitoring field
Monitoring background color	Representation of the background in the monitoring field
Flow control foreground color	Representation of the value in the monitoring fields at the flow control items
Flow control background color	Representation of the background of the monitoring fields at the flow control positions
Number of displayed digits	Number of decimal places in the monitoring field
String length	Maximum length of string variable values in the monitoring field

See also:

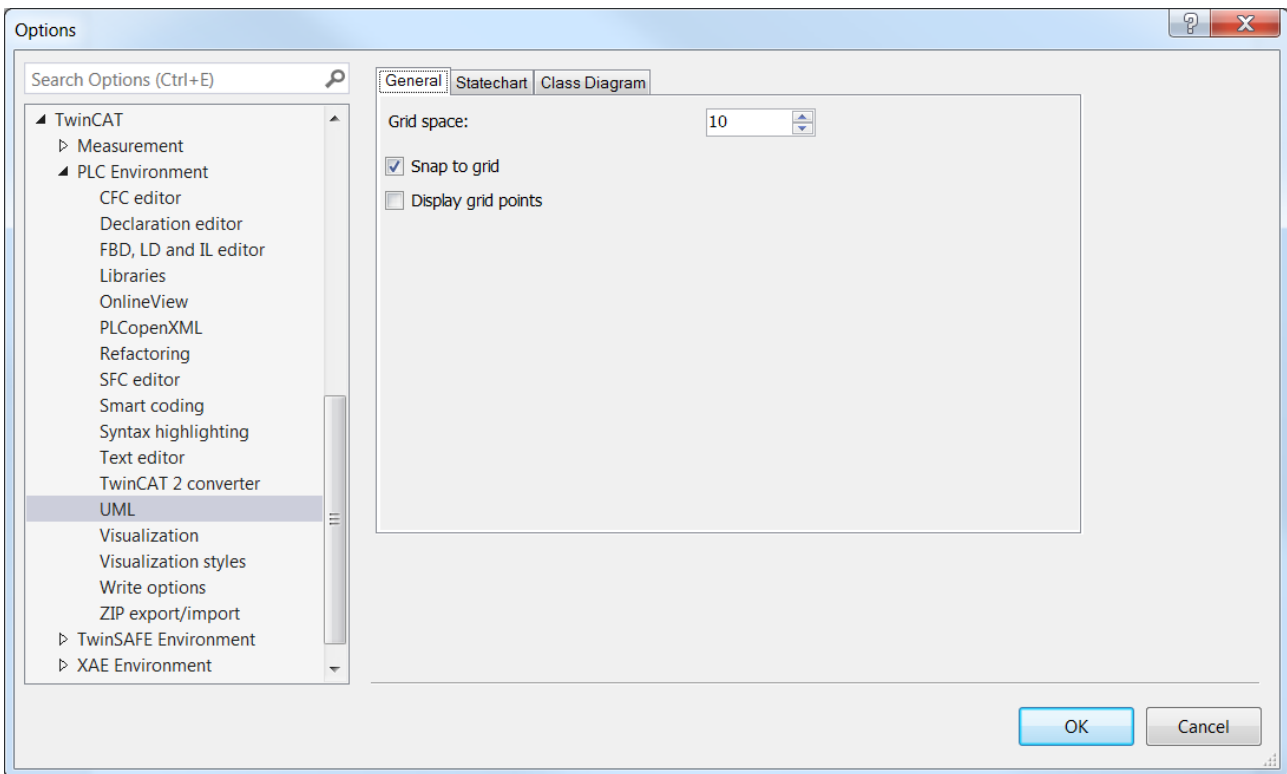
- PLC documentation: [Programming languages and their editors](#) [▶ 622]

17.9.1.13 Dialog Options - UML

Function: The dialog is used to configure the UML editor.

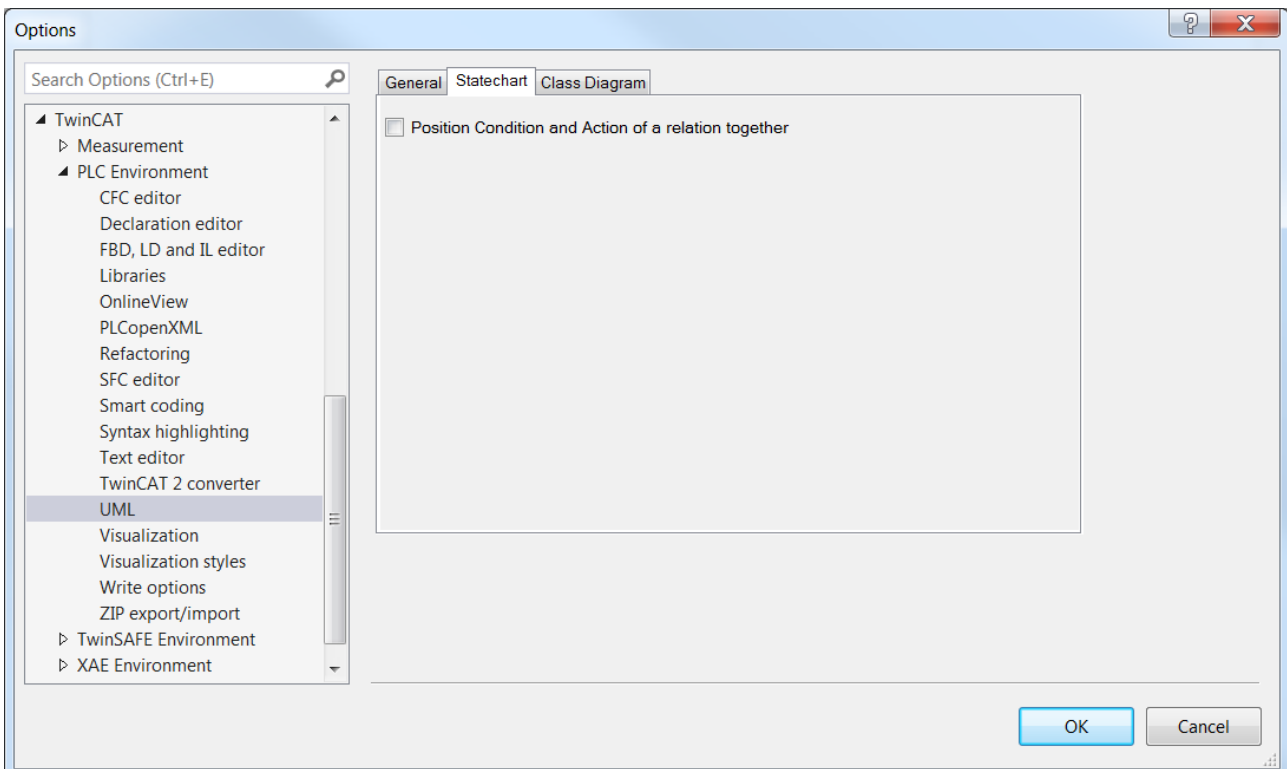
Call: TwinCAT > PLC programming environment > UML

General tab



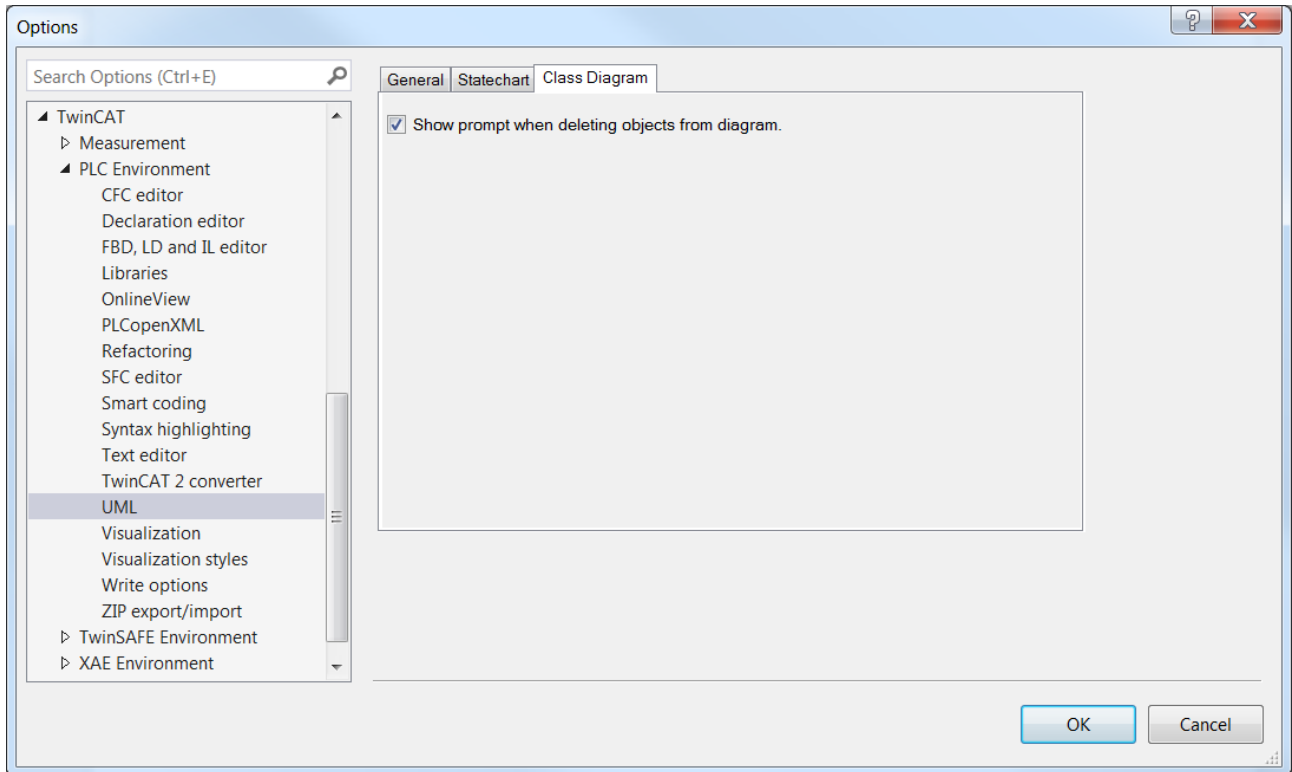
Contact spacing	Grid line spacing in pixels. Default value: 10
Hook onto the grid	<input checked="" type="checkbox"/> : All elements in the UML editors are aligned to the grid.
Display grid points	<input checked="" type="checkbox"/> : The grid points are displayed in the UML editors.

State diagram tab



Position Condition and Action of a relation together	<input checked="" type="checkbox"/> : In the state diagram, a guard condition and an action belonging to the same transition are moved synchronously.
--	---

Class diagram tab



Show prompt when deleting objects from diagram	Objects can be deleted either only from the diagram or from the diagram and from the project. <input type="checkbox"/> : By default, the object is only deleted from the diagram. <input checked="" type="checkbox"/> : When deleting an object, a selection window appears to configure whether the object is to be deleted only from the diagram or from the project.
Skip refactoring preview	<input checked="" type="checkbox"/> : If refactoring is initiated in the diagram, the project-wide change is carried out without first opening the Refactoring dialog with a preview of all change points. See also: <ul style="list-style-type: none"> • PLC documentation: Refactoring [▶ 160]

See also:

- UML documentation: Overview

17.9.1.14 Dialog Options - Visualization

Function: The dialog is used to configure the visualization editor.

Call: TwinCAT > PLC Environment > Visualization

General tab



These settings are only used for integrated visualization, i.e. not for the TwinCAT PLC HMI (TargetVisu) and TwinCAT PLC HMI Web display variants.

Presentation options

Fixed	<input type="radio"/> The visualization retains its original size.
Isotropic	<input type="radio"/> The visualization retains its proportions.
Anisotropic	<input type="radio"/> The visualization adapts to the size of the window in the development system
Antialiased Drawing	<input checked="" type="checkbox"/> The visualization is characterized by antialiasing methods, both during editing and as integrated visualization at runtime. Tip: If a horizontal or vertical line is drawn out of focus on a specific visualization platform, this can currently be corrected by a 0.5 px shift in the direction of the line thickness; see item property Absolute motion , option Use REAL values . Requirement: The platform supports the use of REAL coordinates

Editing options

Linking to shift/key variable	<input checked="" type="checkbox"/> The placeholder <Shift/Key variable> in the visualization element properties is enabled. If you drag an item with the property Color variables, color change into the visualization editor, this property will be configured with the placeholder <Shift/Key variable>. The following items are affected: button, frame, image, line, circular sector, polygon, rectangle, text field, scrollbar
-------------------------------	--

Grid tab

Grid

Visible	<input checked="" type="checkbox"/> Grid lines are visible in the visualization editor at distance size
Active	<input checked="" type="checkbox"/> Grid lines are active in the visualization editor at distance size. The items are aligned with the grid, all position values are on a grid line. An item that is already in a visualization when the grid is enabled is not automatically aligned. To do this, you must first drag it to another position. The grid lines can be active and invisible
Size	Distance of the grid lines in pixels

File options tab

Text file for textual "List components".	File name and location of a file of type .csv. It contains a table with texts in the format of a text list. The entries in the file are provided if the List components function is used as input assistant. You create this file as an export file of the global text list using the Command Import/Export Text Lists.
--	--

See also:

- PLC documentation: [Creating a visualization \[► 376\]](#)

17.9.1.15 Dialog Options - Visualization styles

Function: The dialog is used to configure the visualization styles.

Call: TwinCAT > PLC Environment > visualization styles

Style selection

Display all versions (for experts only)	<input type="checkbox"/> In addition to the currently selected style, all other styles of the repository are available for selection, but only in the latest version. If newer versions are installed for the selected style, they will also be listed. <input checked="" type="checkbox"/> All installed styles in all installed versions are available for selection.
---	--

Style for new Visualization Managers

Last used: <style, version, manufacturer>	Style that is automatically set as selected when you add a new visualization application. It is possible that a display variant is displayed in a different way depending on the device, despite this setting.
Default: <style, version, manufacturer>	Set manufacturer's default style.
<Style, Version, Manufacturer>	Set display variant for style, version and manufacturer.

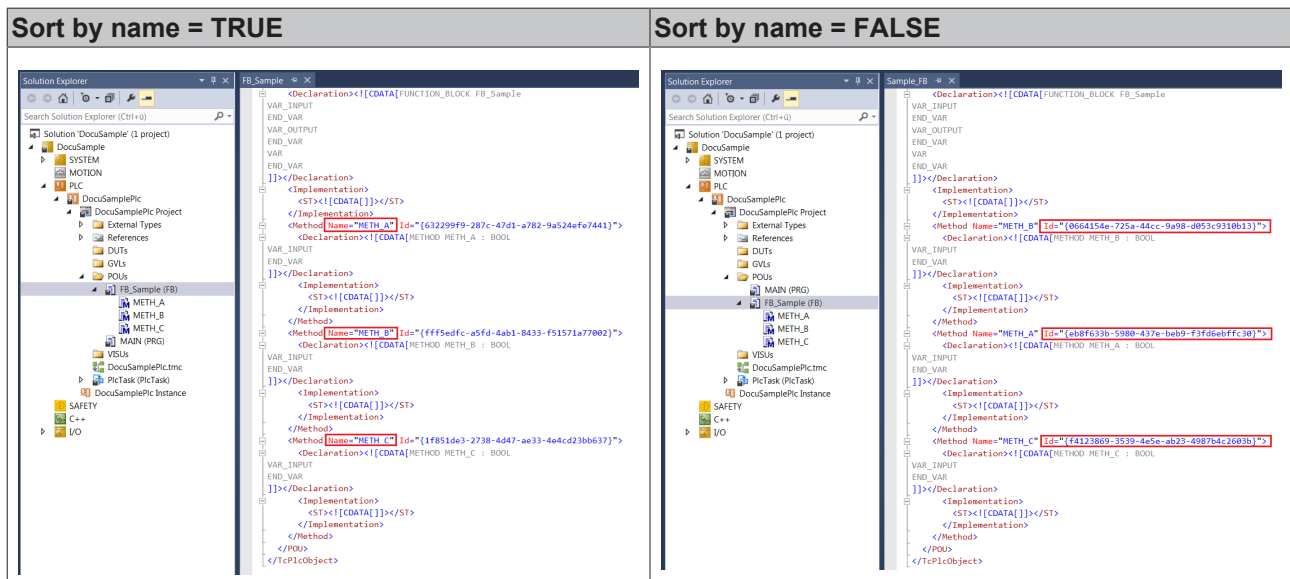
17.9.1.16 Dialog Options - Write Options

Write Options

Separate Line IDs	<ul style="list-style-type: none"> • TRUE: The line IDs of a POU are stored in a separate file (LineIDs.dbg), so that changes in the line IDs do not lead to changes in the POU, which would then be misinterpreted in the source control system as content changes. The requirement for this is from TC3.1 Build 4026 that "Write Line IDs" has the value TRUE. (Default setting: FALSE) Up to TC3.1 Build 4024, line IDs are required for breakpoint handling, for example, and ensure that the code lines can be assigned to machine code instructions.
Sort by name	<ul style="list-style-type: none"> • TRUE (default setting): The subelements of POUs (actions, methods, properties) are sorted by name and not by internal ID (see sample [► 995]).
Write Line IDs	<p>Available from TC3.1 Build 4026</p> <ul style="list-style-type: none"> • TRUE: Line IDs are created and stored for POUs in new projects. (Default setting: FALSE) <p>This setting is the global default setting. When a new PLC project is created, the value of this setting is transferred once to the local project setting. This can be found in the PLC project properties (Category Advanced [► 920]) and can be adapted there for each project.</p>
Write PLC Bookmarks to File	<p>Available from TC3.1 Build 4026</p> <ul style="list-style-type: none"> • TRUE: In new projects, the bookmarks are stored in a separate .bookmarks file in the project directory. (Default setting: FALSE) <p>This setting is the global default setting. When a new PLC project is created, the value of this setting is transferred once to the local project setting. This can be found in the PLC project properties (Category Advanced [► 920]) and can be adapted there for each project.</p>

Sample

The sample illustrates the different storage sequence of the METH_A, METH_B and METH_C methods, depending on whether the **Sort by name** option is enabled or disabled. If the option is disabled (FALSE), the METH_B method is not in second place according to its name, but in first place according to its internal ID.



17.9.1.17 Dialog Options - ZIP export/import

Function: The dialog is used to configure the ZIP export and import settings.

Call: TwinCAT > PLC Environment > ZIP export/import

See also:

- PLC documentation: [Exporting and importing a PLC project \[P 62\]](#)

17.9.2 Command Customize

Function: The command opens the **Customize** dialog. The dialog contains tabs for configuring the user interface. Here you can customize the menus, toolbars and keyboard mapping to suit your individual needs.

Call: Menu Tools

You can restore the TwinCAT standard settings at any time via the **Reset** button.

See also:

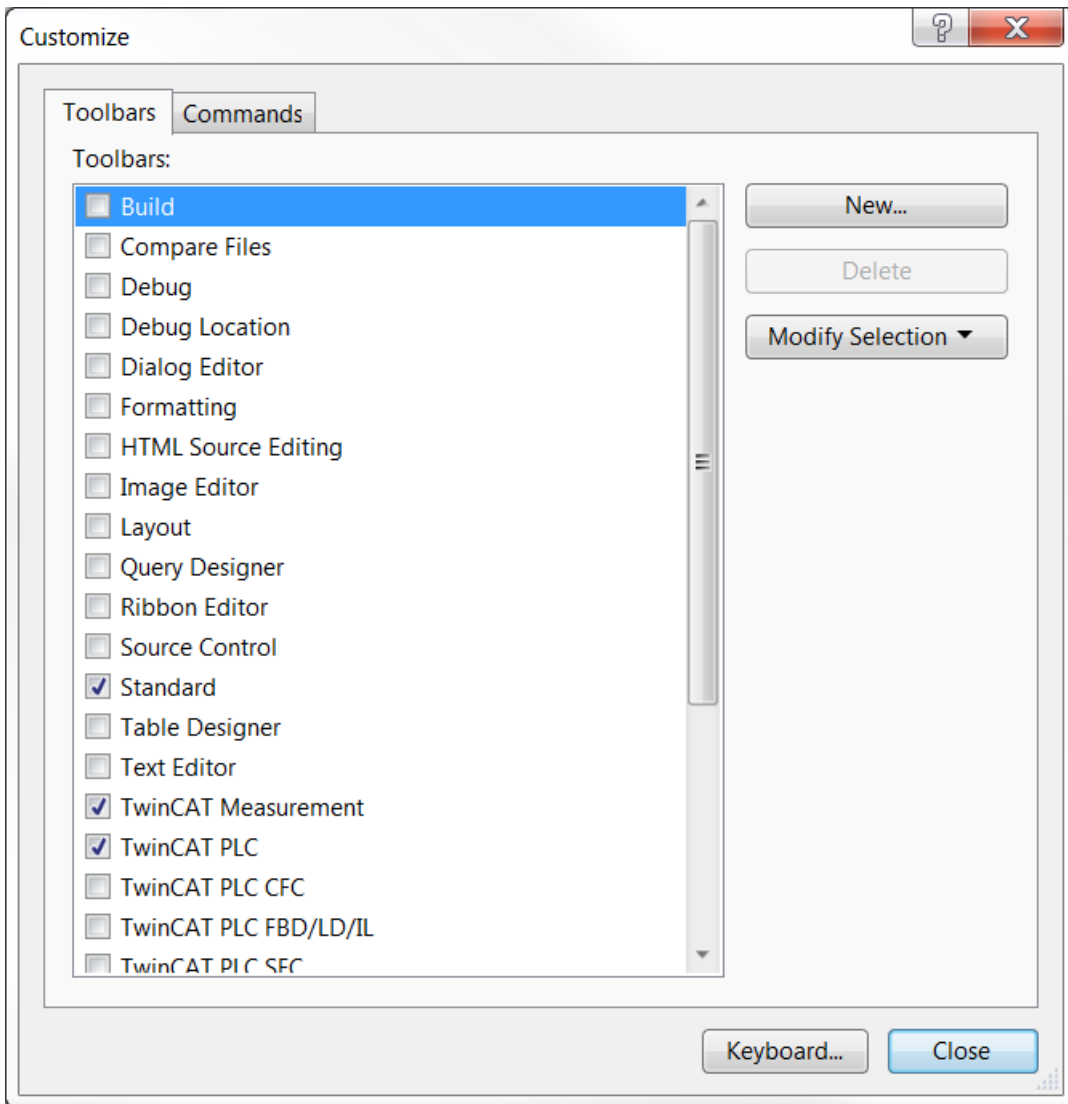
- TC3 User Interface documentation > Customize menus
- TC3 User Interface documentation > Customize toolbars
- TC3 User Interface documentation > Customize keyboard shortcuts

17.9.2.1 Dialog Customize - Toolbars tab

Function: You can use the dialog to create new toolbars or adapt existing toolbars.

Call: Menu Tools > Customize

When you close the dialog with **Close**, the changes become visible in the menu bar of the TwinCAT user interface.



New... (Add toolbar)	TwinCAT adds a toolbar above the selected toolbar. A dialog opens, in which a name can be entered.
Delete (remove toolbar)	TwinCAT removes the selected toolbar. You can only remove toolbars you have created yourself.
Modify Selection (position toolbar)	TwinCAT positions the selected toolbar at the top, bottom, left or right frame of the main window
Keyboard...	Opens the Options dialog, in which you can define keyboard shortcuts.

Toolbars

Displays the currently defined toolbars.	
<input type="checkbox"/> (hide)	Hides the selected toolbar on the user interface.
<input checked="" type="checkbox"/> (show)	Shows the selected hidden toolbar in the TwinCAT user interface.

See also:

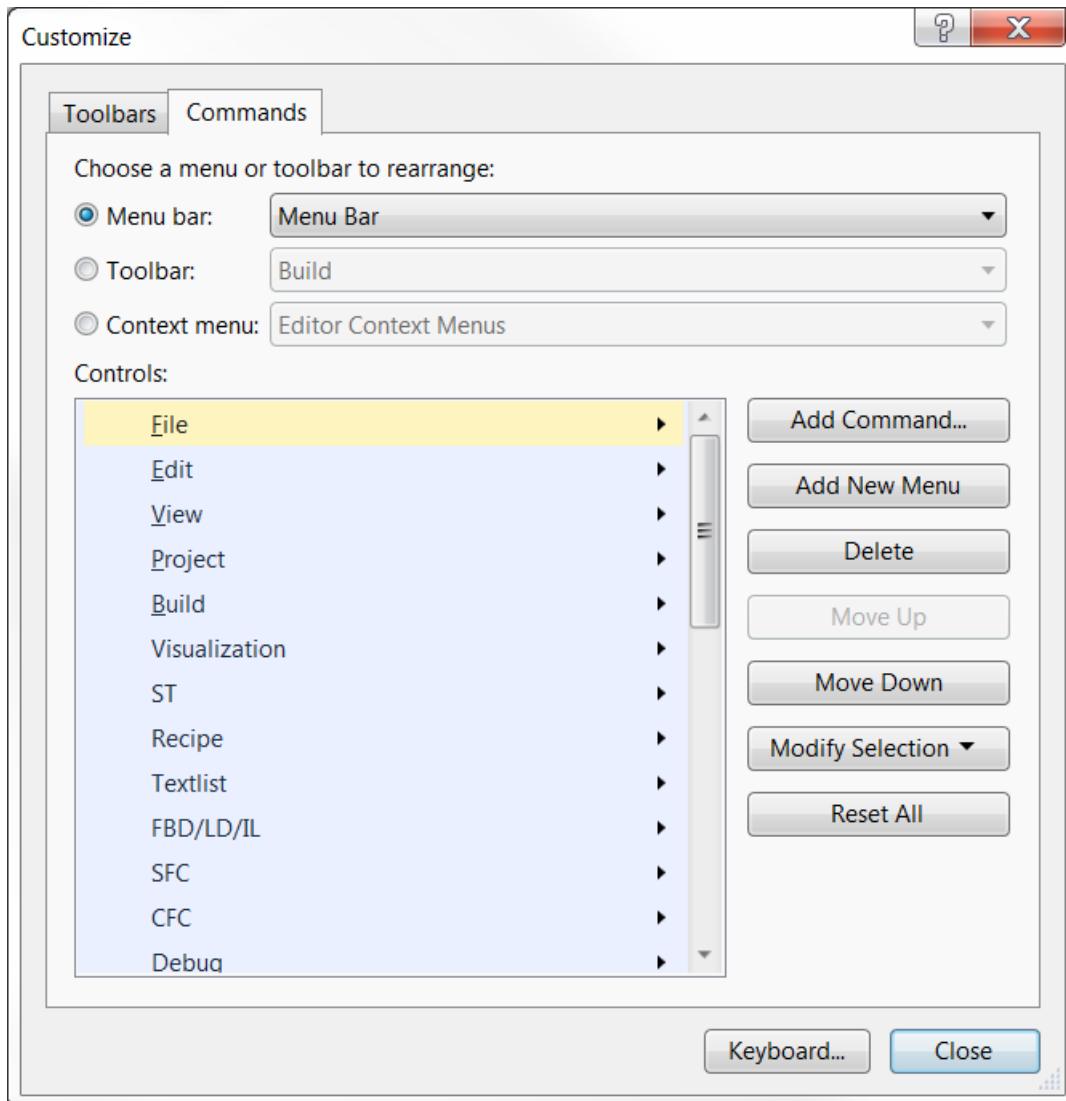
- TC3 User Interface documentation > Customize toolbars

17.9.2.2 Dialog Customize - Commands tab

Function: The dialog allows you to define commands, as well as the structure and content of the menus and toolbars for the user interface.

Call: Menu **Tools** > **Customize**

When you close the dialog with **Close**, the changes become visible in the menu bar of the TwinCAT user interface.



Menus and Toolbars

Display of the currently defined toolbars, menus, submenus and the commands contained therein.	
Menu bar	List of menus and submenus
Toolbar	List of toolbars
Context menu	List of context menus

Controls

Controls	Listing of commands or submenus contained in the selected menu or toolbar. The arrangement from top to bottom corresponds to the arrangement shown later in the TwinCAT menu or in the toolbar.
----------	---

Add Command	Opens the Add Command dialog. The Add Command dialog is used to select one or more commands. Left part: List of categories. Right part: Listing of commands of the selected category. Adds a command above the selected command.
Add New Menu	Adds a new menu above the selected menu.
Modify Selection	Opens a menu in which the name of a newly added menu can be determined.
Delete	Removes the selected menu or command.
Move Up	Moves the selected command or menu up in the order of commands or menus.
Move Down	Moves the selected command or menu down in the order of commands or menus.
Reset All	Resets the entire menu to the default settings.
Keyboard	Opens the Options dialog, in which you can define keyboard shortcuts.

17.10 Window

17.10.1 Command Float

Function: The command detaches a view or window that is docked (fixed) to the frame of the user interface from the frame and places it on the screen as a detached window.

Call: **Window** menu, context menu or button in the header of the view or the tab (window)

The view can then also be placed outside the user interface. Use the **Dock** command to reattach a detached view to the user interface frame.

See also:

- [Command Dock \[► 998\]](#)
- TC3 User Interface documentation: Arranging views and windows

17.10.2 Command Dock

Function: The command "docks" a view, which was previously detached with the **Float** command and is now positioned on the screen as a detached view, back to the user interface frame.

Call: **Window** menu, context menu or button in the header of the view or the tab (window)

See also:

- [Command Float \[► 998\]](#)
- TC3 User Interface documentation: Arranging views and windows


17.10.3 Command Hide

Symbol: 

Function: The command hides a view.

Call: **Window** menu, context menu or button in the header of the view

Requirement: A view is enabled.

Hide means that the view is closed. The command thus corresponds to closing a view via the  button in the header of the view. Use the commands on the **View** menu to unhide or open a hidden view or open.

See also:


- [View \[► 890\]](#)
- TC3 User Interface documentation: Show/hide views

17.10.4 Command Auto Hide All

Function: The command hides all views.

Call: **Window** menu, context menu or button in the header of the view

Hiding means that TwinCAT displays all views only as a tab within the user interface and that it only

becomes visible when you click on the tabs. If you then click the  button in the header bar of the view or select the **Dock** command, the view will be reattached to the user interface.

See also:

- [Command Dock \[► 998\]](#)
- [View \[► 890\]](#)
- TC3 User Interface documentation: Show/hide views

17.10.5 Command Auto Hide


Symbol: 

Function: The command puts a view into the background.

Call: **Window** menu, context menu or button in the header of the view

Requirement: A view is enabled.

Putting in the background means that TwinCAT only displays the view as a tab within the user interface,

which only becomes visible when you click on the tab. If you then click the  button in the header bar again or select the **Dock** command, the view will be reattached to the user interface.

See also:

- [Command Dock \[► 998\]](#)
- TC3 User Interface documentation: Show/hide views

17.10.6 Command Pin tab

Symbol: 

Function: The command attaches the currently active tab to the left margin of the main window.


Call: **Window** menu, context menu or button in the header bar of the tab (window)

Requirement: A tab (window) is enabled.

See also:

- TC3 User Interface documentation: Show/hide views

17.10.7 Command New Horizontal Tab Group

Symbol: 

Function: The command moves the active window to a new, separate tab group below the existing one.

Call: **Window** menu, context menu of the header bar of the tab (window)


Requirement: Several editor windows are arranged next to each other as tabs.

If you open another object in the editor, this is automatically placed in the tab group where the focus is.

See also:

- TC3 User Interface documentation: Arranging views and windows
- [Command New Vertical Tab Group \[► 1000\]](#)

17.10.8 Command New Vertical Tab Group

Symbol: 

Function: The command moves the active window to a new, separate tab group to the right of the existing one.

Call: **Window** menu, context menu of the header bar of the tab (window)

Requirement: Several editor windows are arranged next to each other as tabs.

If you open another object in the editor, this is automatically placed in the tab group where the focus is.

See also:


- TC3 User Interface documentation: Arranging views and windows
- [Command New Horizontal Tab Group \[► 1000\]](#)

17.10.9 Command Reset Window-Layout

Function: This command resets all currently open windows and views to their default positions. You need to confirm the command before it is executed.

Call: **Window** menu

17.10.10 Command Close All Documents

Symbol: 

Function: The command closes all currently open editor windows.

Call: **Window** menu

Requirement: At least one editor window is open.

See also:

- TC3 User Interface documentation: Show/hide views

17.10.11 Command Window

Function: The command opens the **Window** dialog, which displays all open objects. You can activate or close windows in it.

Call: **Window** menu

17.10.12 Window submenu commands

Function: The command activates the selected window.

Call: **Window** menu


For each open editor window the **Window** menu contains a command **<n><Object name>**, through which you can activate the window, i.e. put the focus on it. In offline mode, TwinCAT adds the extension (offline) after the command. For function blocks, the extension (impl) or **<instance path>** is added to distinguish between implementation and instance.

See also:

- [Command Window \[► 1000\]](#)

17.11 SFC

17.11.1 Command Init step

Symbol: 

Function: The command converts the currently selected step to an init step.

Call: Menu **SFC**, context menu

Executing the command changes the frame of the step element to a double line. The step, which previously was the init step, automatically becomes a 'normal' step and is represented with single frame.

The property **Init step** can also be enabled or disabled in the **Properties** view of a step element, although in this case TwinCAT does not automatically adjust the settings of the other steps.

The command can be useful if you want to change the chart. When you create a new SFC object, it automatically includes an init step, followed by a transition (TRUE) and a jump back to the init step.



Note the option to reset the chart to the init step by using the SFC flags SFCInit and SFCReset in online mode.

See also:

- PLC documentation: [Sequential Function Chart \(SFC\) \[► 124\]](#)
- PLC documentation: [SFC Editor \[► 635\]](#)
- PLC documentation: [SFC element properties \[► 650\]](#)

17.11.2 Command Insert step transition

Symbol: 

Function: The command adds a step and a transition before the currently selected position.

Call: Menu **SFC**, context menu

If you have selected a step, TwinCAT inserts a new step transition combination. If you have selected a transition, a new transition step combination is inserted.

The new step is called step<n> by default. n is a sequential number, starting with 0 for the first step, which is added in addition to the init step. Accordingly, the new transition is called Trans<n> by default. You can edit the default names directly by clicking on the name.

See also:

- [Command Insert step-transition after](#) [► 1002]
- PLC documentation: [Sequential Function Chart \(SFC\)](#) [► 124]
- PLC documentation: [SFC Editor](#) [► 635]
- PLC documentation: [SFC elements step and transition](#) [► 644]

17.11.3 Command Insert step-transition after

Symbol: 

Function: The command adds a step and a transition after the currently selected position.

Call: Menu **SFC**, context menu


If you have selected a step, TwinCAT inserts a new transition step combination. If you have selected a transition, a new step transition combination is inserted.

The new step is called Step<n> by default. n is a sequential number, starting with 0 for the first step, which is added in addition to the init step. Accordingly, the new transition is called Trans<n> by default. You can edit the default names directly by clicking on the name.

See also:

- [Command Insert step transition](#) [► 1001]
- PLC documentation: [Sequential Function Chart \(SFC\)](#) [► 124]
- PLC documentation: [SFC Editor](#) [► 635]
- PLC documentation: [SFC Elements 'Step' and 'Transition'](#)

17.11.4 Command Parallel

Symbol: 

Function: The command converts the selected alternative branch to a parallel branch.

Call: Menu **SFC**, context menu

Requirement: The horizontal connecting line of a branch is selected.

Note that after converting a branch, you must check and adjust the sequence of steps and transitions before and after the branch.

See also:

- [Command Alternative](#) [► 1002]
- PLC documentation: [Sequential Function Chart \(SFC\)](#) [► 124]
- PLC documentation: [SFC Editor](#) [► 635]

17.11.5 Command Alternative

Symbol: 

Function: The command converts the selected parallel branch to an alternative branch.

Call: Menu **SFC**, context menu


Requirement: The horizontal connecting line of a branch is selected.

Note that after converting a branch, you must check and adjust the sequence of steps and transitions before and after the branch.

See also:

- [Command Parallel](#) [► 1002]
- PLC documentation: [Sequential Function Chart \(SFC\)](#) [► 124]
- PLC documentation: [SFC Editor](#) [► 635]

17.11.6 Command Insert Branch

Symbol: 

Function: The command adds a branch to the left of the currently selected position.


Call: Menu **SFC**, context menu

The behavior of the command corresponds to the command **Insert branch right**.

See also:

- [Command Insert branch right](#) [► 1003]
- PLC documentation: [Sequential Function Chart \(SFC\)](#) [► 124]
- PLC documentation: [SFC Editor](#) [► 635]
- PLC documentation: [SFC element Branch](#) [► 647]

17.11.7 Command Insert branch right

Symbol: 

Function: The command adds a branch to the right of the currently selected position.

Call: Menu **SFC**, context menu

The type of the inserted branch depends on the selected element:

- If the top-most element of the currently selected elements is a transition or an alternative branch, TwinCAT inserts an alternative branch.
- If the top-most element of the currently selected element is a step, a macro, a jump or a parallel branch, TwinCAT adds a parallel branch with label Branch<x>, where x is a sequential number. You can edit this default label name. You can specify the label as the target of a jump.
- If a common element of an existing branch is currently selected (horizontal line), TwinCAT adds the new branch at the far right as an additional branch. If an entire branch of an existing branch is currently selected, TwinCAT adds the new branch directly to the right of it as a new branch.



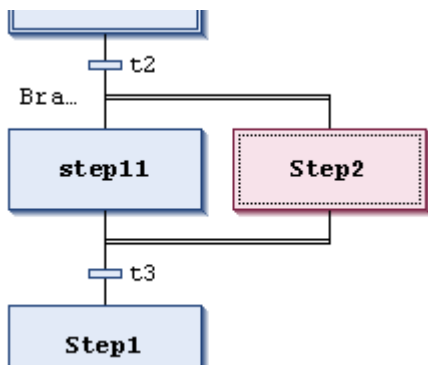
Note that you can use the commands **Alternative** or **Parallel** to convert a branch to the respective other type.

Example: Parallel branch

The following diagram shows a newly inserted parallel branch, generated with Command Insert branch right, while step11 was selected. TwinCAT automatically inserts a step (Step2 in the example).

Processing in online mode: If t2 returns TRUE, TwinCAT executes Step2 immediately after step11, before t3 is evaluated.

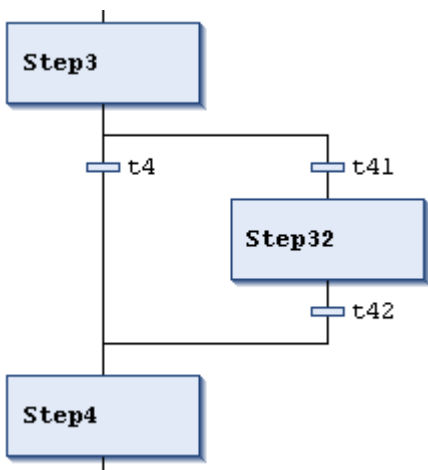
In contrast to alternative branches, TwinCAT therefore executes both branches.



Example: Alternative branch

The following diagram shows a newly inserted alternative branch, generated with Command Insert branch right, while transition t4 was selected. TwinCAT automatically inserts a step (Step32 in the example), a preceding and a subsequent transition (t41, t42).

Processing in online mode: If Step3 is active, TwinCAT evaluates the following transitions (t4, t41) from left to right. The first junction of the branch in which the first transition returns TRUE is executed. In contrast to parallel branching, only one junction is executed.



See also:

- [Command Alternative \[► 1002\]](#)
- [Command Parallel \[► 1002\]](#)
- [Command Insert Branch \[► 1003\]](#)
- PLC documentation: [Sequential Function Chart \(SFC\) \[► 124\]](#)
- PLC documentation: [SFC Editor \[► 635\]](#)
- PLC documentation: [SFC element Branch \[► 647\]](#)

17.11.8 Command Insert action association

Symbol:

Function: The command assigns an IEC action to a step.

Call: Menu **SFC**, context menu

Requirement: A step is selected.

TwinCAT adds the action element to the right of the currently selected step element.

If you have already assigned one or more actions to the step, they are displayed in a "action list", one below the other. The new action is then placed as follows:

- As the first action of the step, i.e. at the top of the action list, if you have selected the step element.
- Directly before, i.e. above the action, if you have selected one of the existing actions in the action list of the step.


The left part of the action element contains the qualifier, by default N, in the right part you enter the action name. To do this, click in the box to get an editing frame. You must have created this action as a POU in the project.

You can also edit the qualifier. A list of the valid qualifiers is described in section "Qualifiers for actions in SFC".

See also:

- [Command Insert action association after \[► 1005\]](#)
- PLC documentation: [Sequential Function Chart \(SFC\) \[► 124\]](#)
- PLC documentation: [SFC Editor \[► 635\]](#)
- PLC documentation: [Qualifier for actions in SFC \[► 637\]](#)

17.11.9 Command Insert action association after

Symbol: 

Function: The command assigns an IEC action to a step.

Call: Menu **SFC**, context menu

Requirement: A step is selected.

The command corresponds to the description of Insert action association. The difference is that TwinCAT places the new action not at the first, but at the last position of the action list. If you have selected an action in the action list, TwinCAT does not place the new action above it, but below it.

See also:

- [Command Insert action association \[► 1004\]](#)
- PLC documentation: [Sequential Function Chart \(SFC\) \[► 124\]](#)
- PLC documentation: [SFC Editor \[► 635\]](#)
- PLC documentation: [Qualifier for actions in SFC \[► 637\]](#)

17.11.10 Command Insert jump

Symbol: 

Function: The command inserts a jump element before the currently selected element.

Call: Menu **SFC**, context menu


Requirement: A step is selected.

TwinCAT automatically inserts the jump with jump target step. You must then replace this jump target with a real jump target. You can select the target with the Input Assistant.

See also:

- [Command Insert jump after \[► 1006\]](#)
- PLC documentation: [Sequential Function Chart \(SFC\) \[► 124\]](#)
- PLC documentation: [SFC Editor \[► 635\]](#)
- PLC documentation: [SFC element Jump \[► 649\]](#)

17.11.11 Command Insert jump after

Symbol: 

Function: The command inserts a jump element after the currently selected element.

Call: **SFC** menu

TwinCAT automatically inserts the jump with jump target step. You must then replace this jump target with a real jump target. You can select the target with the Input Assistant.

See also:

- [Command Insert jump](#) [▶ 1005]
- PLC documentation: [Sequential Function Chart \(SFC\)](#) [▶ 124]
- PLC documentation: [SFC Editor](#) [▶ 635]
- PLC documentation: [SFC element Jump](#) [▶ 649]

17.11.12 Command Insert macro

Symbol: 

Function: The command inserts a macro element before the currently selected element.

Call: Menu **SFC**, context menu


The new macro is called Macro<x> by default. x is a sequential number, starting with 0 for the first macro. You can edit the default name directly by clicking on the name.

To edit the macro, open it with the command **Show macro** in the macro editor.

See also:

- [Command Show macro](#) [▶ 1007]
- [Command Insert macro after](#) [▶ 1006]
- PLC documentation: [Sequential Function Chart \(SFC\)](#) [▶ 124]
- [SFC Editor](#) [▶ 635]

17.11.13 Command Insert macro after

Symbol: 

Function: The command inserts a macro element after the currently selected element.

Call: Menu **SFC**, context menu

The command corresponds to the description of the command **Insert macro**.

See also:

- [Command Insert macro](#) [▶ 1006]
- [Command Show macro](#) [▶ 1007]
- PLC documentation: [Sequential Function Chart \(SFC\)](#) [▶ 124]
- PLC documentation: [SFC Editor](#) [▶ 635]

17.11.14 Command Show macro

Symbol: 

Function: The command opens a macro in the macro editor for editing.

Call: SFC menu

Requirement: A macro is selected.


The command causes TwinCAT to close the main view of the SFC editor and open the macro editor instead. This is also an SFC editor, in which you can now edit the part of the SFC diagram that is displayed as a macro box in the main view.

Use the command **Exit macro** to return to the main view.

See also:

- [Command Exit macro \[► 1007\]](#)
- PLC documentation: [Sequential Function Chart \(SFC\) \[► 124\]](#)
- PLC documentation: [SFC Editor \[► 635\]](#)

17.11.15 Command Exit macro

Symbol: 

Function: The command closes the macro editor and returns to the main view of the SFC editor.

Call: SFC menu

Requirement: A macro is open in the macro editor.

See also:

- [Command Show macro \[► 1007\]](#)
- PLC documentation: [Sequential Function Chart \(SFC\) \[► 124\]](#)
- PLC documentation: [SFC Editor \[► 635\]](#)

17.11.16 Command Insert after

Symbol: 

Function: The command inserts the elements from the clipboard after the currently selected position.

Call: SFC menu

17.11.17 Command Add entry action

Symbol: 

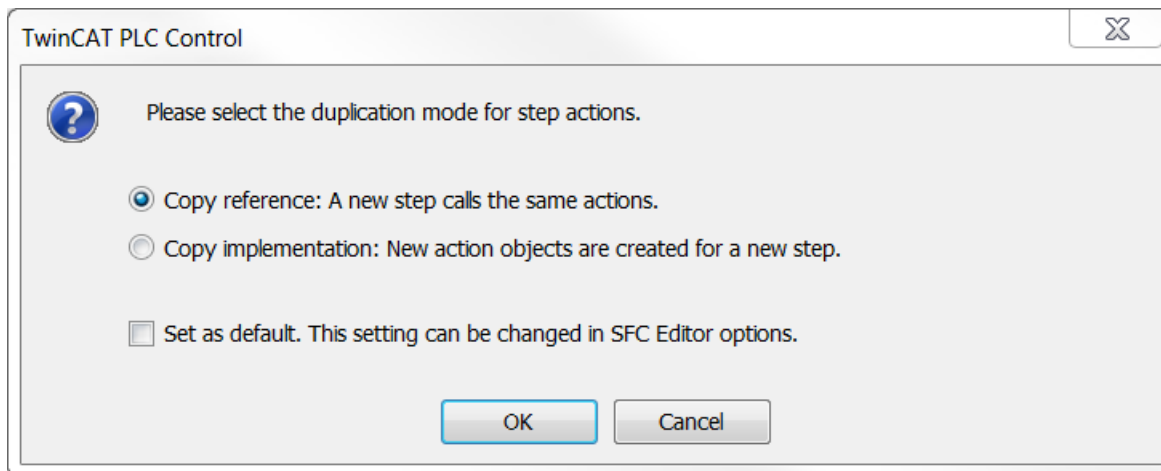
Function: The command leads to the **Add entry action** dialog, in which you define a new step action of type "entry action". Depending on the SFC options, a prompt for selecting the duplication mode for the new step action may appear beforehand.

Call: SFC menu, context menu of the selected step element

Requirement: A step element is selected.

The entry action is automatically opened in the ST editor. The step element is assigned an "E" in the lower left corner.

Query dialog for selecting the duplication mode



Copy reference. A new step will call the same actions	If the step is copied in the SFC, the link to the step action(s) is also copied. The steps copied from one another will therefore call all the same actions.
Copy implementation. New action objects are created for the new step	This means "embedding" of the step actions for the copied steps. By default, the newly created action objects appear below the SFC function block in the PLC project tree in the Solution Explorer. Initially, these objects contain a copy of the original implementation code for the respective action.
Set as default. This setting can be changed in the SFC editor options.	The settings in the dialog are accepted as the default setting. You can change the default setting in the TwinCAT options in the category SFC editor . To do this, in the group field Step actions in the drop-down list Standard insert method select the entry Always ask, Copy reference or Duplicate implementation .

See also:

- Command Options > [Dialog Options - SFC editor](#) [▶ 977]
- PLC documentation: [Command Add exit action](#) [▶ 1008]
- PLC documentation: [Sequential Function Chart \(SFC\)](#) [▶ 124]
- PLC documentation: [SFC Editor](#) [▶ 635]
- PLC documentation: [Programming in Sequential Function Chart \(SFC\)](#) [▶ 124]
- PLC documentation: [SFC element Action](#) [▶ 645]

17.11.18 Command Add exit action

Symbol:

Function: The command leads to the **Add exit action** dialog, in which you define a new step action of type entry action. Depending on the SFC options, a prompt for selecting the duplication mode for the new step action may appear beforehand. Please refer to the help page for the command **Add entry action**.

Call: **SFC** menu, context menu of the selected step element

Requirement: A step element in the SFC is selected.

See also:

- [Command Add entry action](#) [▶ 1007]
- Command Options > [Dialog Options - SFC editor](#) [▶ 977]

- PLC documentation: [Sequential Function Chart \(SFC\) \[► 124\]](#)
- PLC documentation: [Programming in Sequential Function Chart \(SFC\) \[► 124\]](#)
- PLC documentation: [SFC Editor \[► 635\]](#)
- PLC documentation: [SFC element Action \[► 645\]](#)

17.11.19 Command Change duplication - Set

Function: The command permanently links each step action or transition that is called by a step or a transition in the SFC function block to the caller. Thus the action or transition object can then only be called by exactly this one caller (pseudo-embedding). As a result, copying step and transition elements that call actions or transitions automatically creates new action or transition objects. The implementation code is copied in each case.

Call: SFC menu

For details of the duplication mode, see the help page for the SFC element properties and the instructions for adding step actions.

See also:

- PLC documentation: [SFC Element Properties \[► 650\]](#)
- PLC documentation: [Programming in Sequential Function Chart \(SFC\) \[► 124\]](#)

17.11.20 Command Change duplication - Remove

Function: The command removes the fixed link of action or transition objects with the step or transition that it calls for the entire SFC function block. This cancels the pseudo-embedding of the action or transition objects. If step and transition elements that call actions or transitions are then copied, the copies call the same actions and transitions as the source.

Call: SFC menu

For details of the duplication mode, see the help page for the SFC element properties and the instructions for adding step actions.

See also:

- PLC documentation: [SFC Element Properties \[► 650\]](#)
- PLC documentation: [Programming in Sequential Function Chart \(SFC\) \[► 124\]](#)

17.11.21 Command Insert step



The command is not included as standard in the SFC menu.

Symbol:

Function: The command adds a step before the currently selected position.

Call: SFC menu, Context menu in the SFC editor

The new step is called Step<n> by default. n is a sequential number, starting with 0 for the first step, which is added in addition to the init step. The name can be edited by clicking on it.

Inserting a step without a transition or a transition without a step results in a compile error.

See also:

- [Command Insert step-transition after \[► 1002\]](#)

- [Command Init step \[► 1001\]](#)
- PLC documentation: [SFC elements step and transition \[► 644\]](#)

17.11.22 Command Insert step after



The command is not included as standard in the **SFC** menu.

Symbol:

Function: The command inserts a step after the currently selected position.

Call: **SFC** menu, Context menu in the SFC editor

The new step is called Step<n> by default. n is a sequential number, starting with 0 for the first step, which is added in addition to the init step. The name can be edited by clicking on it.

Inserting a step without a transition or a transition without a step results in a compile error.

See also:

- [Command Init step \[► 1001\]](#)
- [Command Insert step-transition after \[► 1002\]](#)
- PLC documentation: [SFC element Jump \[► 649\]](#)

17.11.23 Command Insert transition



The command is not included as standard in the **SFC** menu.

Symbol:

Function: The command adds a transition before the currently selected position.

Call: **SFC** menu, Context menu in the SFC editor

The new transition is called Trans<n> by default. n is a sequential number, starting with 0 for the first transition. The name can be edited by clicking on it.

Inserting a step without a transition or a transition without a step results in a compile error.

See also:

- [Command Insert step-transition after \[► 1002\]](#)
- PLC documentation: [SFC elements step and transition \[► 644\]](#)

17.11.24 Command Insert transition after



The command is not included as standard in the **SFC** menu.

Symbol:

Function: The command inserts a transition after the currently selected position.

Call: **SFC** menu, Context menu in the SFC editor

The new transition is called Trans<n> by default. n is a sequential number, starting with 0 for the first transition. The name can be edited by clicking on it.

Inserting a step without a transition or a transition without a step results in a compile error.

See also:

- [Command Insert step after \[▶ 1010\]](#)
- PLC documentation: [SFC elements step and transition \[▶ 644\]](#)

17.12 CFC

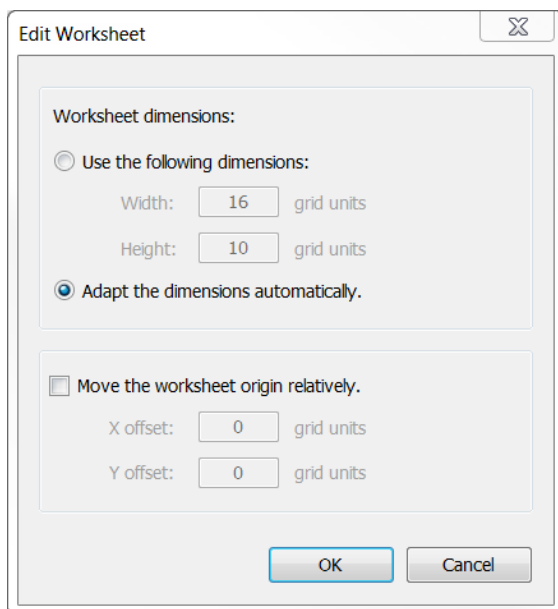
17.12.1 Command Edit Worksheet

Function: The command opens the **Edit Worksheet** dialog, in which you specify the size of the worksheet.

Call: CFC menu

Requirements: A CFC editor is active.

Edit Worksheet dialog



Use the following dimensions	Here, you set the size of the worksheet. Your change is only be accepted if the size is sufficient for the existing program.
Adapt the dimensions automatically	Automatically adapts the size of the worksheet to the size of your program.
Move the worksheet origin relatively	Moves the worksheet on the x or y axis. Entering negative numbers is allowed.

See also:

- PLC documentation: [Continuous Function Chart \(CFC\) \[▶ 113\]](#)
- PLC documentation: [CFC Editor \[▶ 666\]](#)


17.12.2 Command Edit page size

Function: The command opens the Edit Page Size dialog. This can be used to change the size of the page-based CFC editor.

Call: CFC menu

Requirements: A page-oriented CFC editor is active.


Dialog Edit Page Size

Width	Width of the page (minimum 24, maximum 1024). Elements outside the working area are highlighted in red.
Height	Page height (minimum 24, maximum 1024). Elements outside the working area are highlighted in red.
Border width	Margin width (minimum 6, maximum 25% or page width).
Set as default for new CFC objects	 : The current settings are set as default for new CFC objects.

See also:

- PLC documentation: [Continuous Function Chart \(CFC\) \[► 113\]](#)
- PLC documentation: [CFC Editor \[► 666\]](#)
- PLC documentation: [CFC element Page \[► 676\]](#)

17.12.3 Command Negate

Symbol: 

Function: The command negates the selected function block input or output.


Call: CFC menu, context menu

Requirements: A CFC editor is active. A function block input or output is selected.

See also:

- PLC documentation: [Continuous Function Chart \(CFC\) \[► 113\]](#)
- PLC documentation: [CFC Editor \[► 666\]](#)

17.12.4 Command EN/ENO

Symbol: 

Function: The command adds a Boolean input EN (Enable) and a Boolean output ENO (Enable Out) to the selected function block.

Call: CFC menu, context menu


Requirements: A CFC editor is active. A function block is selected.

The added input "EN" enables the function block. The function block is only executed if it has the value TRUE. The value of this signal is output at output ENO.

See also:

- PLC documentation: [Continuous Function Chart \(CFC\) \[► 113\]](#)
- PLC documentation: [CFC Editor \[► 666\]](#)

17.12.5 Command None

Symbol: 

Function: The command removes a reset (R), a set (S) or a REF from the input of the "output" element.


Call: CFC > Set/Reset menu; context menu > Set/Reset

Requirements: A CFC editor is active. The input of an **Output** element is selected.

See also:

- PLC documentation: [Continuous Function Chart \(CFC\) \[► 113\]](#)
- PLC documentation: [CFC Editor \[► 666\]](#)

17.12.6 Command R (reset)

Symbol: 

Function: The command adds a reset to the input of a Boolean element **Output**.

Call: Menu CFC > Set Reset, context menu > Set/Reset


Requirements: A CFC editor is active. The input of an "Output" element is selected.

If an **Output** element has a reset input, the Boolean output value is set to FALSE when the value of the input is TRUE. The value FALSE at the output is retained, even if the input value changes again.

See also:

- [Command S \(set\) \[► 1013\]](#)
- PLC documentation: [Continuous Function Chart \(CFC\) \[► 113\]](#)
- PLC documentation: [CFC Editor \[► 666\]](#)

17.12.7 Command S (set)

Symbol: 

Function: The command adds a set (S) to the input of a Boolean element "Output".

Call: Menu CFC > Set/Reset, context menu > Set/Reset


Requirements: A CFC editor is active. The input of an **Output** element is selected.

If an **Output** element has a set input, the Boolean output value is set to TRUE when the value of the input is TRUE. The value TRUE at the output is retained, even if the input value changes again.

See also:

- [Command R \(reset\) \[► 1013\]](#)
- PLC documentation: [Continuous Function Chart \(CFC\) \[► 113\]](#)
- PLC documentation: [CFC Editor \[► 666\]](#)

17.12.8 Command REF = (reference assignment)

Symbol: 

Function: The command assigns a reference to an "Output" element.

Call: Menu CFC > Set/Reset, context menu > Set/Reset

Requirements: A CFC editor is active. The input of an **Output** element is selected.

Example:

Declaration:

```
refInt : REFERENCE TO INT;
nVar1 : INT;
```

CFC:



This corresponds to the ST code

```
refInt REF= nVar1;
```

Further information can be found in the description of the REFERENCE TO data type.

See also:

- PLC documentation: [Continuous Function Chart \(CFC\) \[► 113\]](#)
- PLC documentation: [CFC Editor \[► 666\]](#)
- PLC documentation: [Reference \[► 770\]](#)

17.12.9 Command Display Execution Order

Symbol:

Function: The command briefly displays a mark with number for all CFC elements of the programming object.

Call:

- **CFC > Execution order** menu
- Context menu in the CFC editor > **Execution order**

Requirements: A CFC editor is active and the Automatic data flow mode is enabled.

The numbers reflect the automatically determined execution order. The execution order is determined according to data flow and, in case of multiple networks, according to their topological position in the editor.

As soon as you click in the CFC editor, the marks are hidden.

See also:

- PLC documentation: [Continuous Function Chart \(CFC\) \[► 113\]](#)
- PLC documentation: [CFC Editor \[► 666\]](#)
- PLC documentation: [Automatic execution order by data flow \[► 120\]](#)
- PLC documentation: [CFC settings \[► 905\]](#)

17.12.10 Command Set Start of Feedback


Symbol:

Function: The command defines the selected element as the starting point within a feedback loop.

Call:

- **CFC > Execution order** menu
- Context menu in the CFC editor > **Execution order**


Requirements: A CFC editor is active and the Automatic data flow mode is enabled. A network of the CFC programming block maintains a feedback loop and an element within this feedback loop is selected.

In the CFC editor, the starting point within feedbacks is decorated with the symbol . This element has the lowest execution order number within the feedbacks. At runtime, the processing of the feedback starts with this element.

See also:

- PLC documentation: [Command Display Execution Order \[▶ 1014\]](#)
- PLC documentation: [Continuous Function Chart \(CFC\) \[▶ 113\]](#)
- PLC documentation: [CFC Editor \[▶ 666\]](#)
- PLC documentation: [Automatic execution order by data flow \[▶ 120\]](#)
- PLC documentation: [CFC settings \[▶ 905\]](#)

17.12.11 Command Move to Beginning

Symbol: 

Function: The command numbers the elements so that the selected elements are at the beginning of the execution order.

Call: CFC > Execution order menu, context menu > Execution order


Requirements: A CFC editor is active and the Explicit Execution Order mode is enabled. At least one element is selected.

The selected elements are assigned the lowest numbers starting with 0 while keeping the previous order. The remaining elements are numbered so that their execution order remains. The topological positions of the elements are not changed.

See also:

- PLC documentation: [Continuous Function Chart \(CFC\) \[▶ 113\]](#)
- PLC documentation: [CFC Editor \[▶ 666\]](#)
- PLC documentation: [Automatic execution order by data flow \[▶ 120\]](#)
- PLC documentation: [CFC settings \[▶ 905\]](#)

17.12.12 Command Move to End

Symbol: 

Function: The command numbers the elements so that the selected elements are at the end of the execution order.

Call: CFC > Execution order menu, context menu > Execution order


Requirements: A CFC editor is active and the Explicit Execution Order mode is enabled. At least one element is selected.

The selected elements are assigned the highest numbers while keeping the previous order. The remaining elements are numbered so that their execution order remains. The topological positions of the elements are not changed.

See also:

- PLC documentation: [Continuous Function Chart \(CFC\) \[▶ 113\]](#)
- PLC documentation: [CFC Editor \[▶ 666\]](#)
- PLC documentation: [Automatic execution order by data flow \[▶ 120\]](#)
- PLC documentation: [CFC settings \[▶ 905\]](#)

17.12.13 Command Forward by one

Symbol: 

Function: The command numbers the elements so that the selected elements are one ahead.

Call: CFC > Execution order menu, context menu > Execution order


Requirements: A CFC editor is active and the Explicit Execution Order mode is enabled. At least one element is selected.

The selected elements are numbered one less while retaining the previous order, so that they are processed one execution position earlier. The remaining elements are numbered so that their execution order remains. The topological positions of the elements are not changed.

See also:

- PLC documentation: [Continuous Function Chart \(CFC\) \[► 113\]](#)
- PLC documentation: [CFC Editor \[► 666\]](#)
- PLC documentation: [Automatic execution order by data flow \[► 120\]](#)
- PLC documentation: [CFC settings \[► 905\]](#)

17.12.14 Command Back by one

Symbol: 

Function: The command numbers the elements so that the selected elements are one behind.

Call: CFC > Execution order menu, context menu > Execution order

Requirements: A CFC editor is active and the Explicit Execution Order mode is enabled. At least one element is selected.

The selected elements are numbered one higher while retaining the previous order, so that they are processed one execution position later. The remaining elements are numbered so that their execution order remains. The topological positions of the elements are not changed.

See also:

- PLC documentation: [Continuous Function Chart \(CFC\) \[► 113\]](#)
- PLC documentation: [CFC Editor \[► 666\]](#)
- PLC documentation: [Automatic execution order by data flow \[► 120\]](#)
- PLC documentation: [CFC settings \[► 905\]](#)

17.12.15 Command Set Execution Order

Function: The command opens a dialog for setting the number of the selected element to any value.

Call: CFC > Execution order menu, context menu > Execution order

Requirements: A CFC editor is active and the Explicit Execution Order mode is enabled. Exactly one element is selected.

The selected element gets the number specified in the dialog. The remaining elements are numbered so that their execution order remains. The topological positions of the elements are not changed.

See also:

- PLC documentation: [Command Order by Data Flow \[► 1017\]](#)
- PLC documentation: [Command Order By Topology \[► 1017\]](#)

- PLC documentation: [Continuous Function Chart \(CFC\) \[► 113\]](#)
- PLC documentation: [CFC Editor \[► 666\]](#)
- PLC documentation: [Automatic execution order by data flow \[► 120\]](#)
- PLC documentation: [CFC settings \[► 905\]](#)

17.12.16 Command Order by Data Flow

Function: The command numbers the elements in the program according to data flow and, in case of several networks, according to their topological position in the editor.

Call: **CFC > Execution order** menu, context menu > Execution order

Requirements: A CFC editor is active and the Explicit Execution Order mode is enabled.

The execution order is arranged according to the data flow and (in case of multiple networks) according to the topological position of the networks. All numbered elements of the programming block are set accordingly. The execution order is then identical to that in automatic data flow mode. The topological positions of the elements are not changed.

See also:

- PLC documentation: [Command Order By Topology \[► 1017\]](#)
- PLC documentation: [Command Set Execution Order \[► 1016\]](#)
- PLC documentation: [Continuous Function Chart \(CFC\) \[► 113\]](#)
- PLC documentation: [CFC Editor \[► 666\]](#)
- PLC documentation: [Automatic execution order by data flow \[► 120\]](#)
- PLC documentation: [CFC settings \[► 905\]](#)

17.12.17 Command Order By Topology

Function: The command arranges the execution order of the elements according to their topological position from right to left and top to bottom.

Call: **CFC > Execution order** menu, context menu > Execution order

Requirements: A CFC editor is active and the Explicit Execution Order mode is enabled. At least one element is selected.

The command affects all elements in the program, even if not all elements are selected when the command is executed. The topological positions of the elements are not changed.

See also:

- PLC documentation: [Command Order by Data Flow \[► 1017\]](#)
- PLC documentation: [Command Set Execution Order \[► 1016\]](#)
- PLC documentation: [Continuous Function Chart \(CFC\) \[► 113\]](#)
- PLC documentation: [CFC Editor \[► 666\]](#)
- PLC documentation: [Automatic execution order by data flow \[► 120\]](#)
- PLC documentation: [CFC settings \[► 905\]](#)

17.12.18 Command Connect Selected Pins

Symbol: 

Function: The command establishes a link between the selected pins.

Call: CFC menu, context menu


Requirements: A CFC editor is active. Exactly one output and several inputs are selected.

To select the pins, keep **[CTRL]** pressed while clicking the pins. Then execute the command.

See also:

- [Command Select Connected Pins \[▶ 1018\]](#)
- PLC documentation: [Continuous Function Chart \(CFC\) \[▶ 113\]](#)
- PLC documentation: [CFC Editor \[▶ 666\]](#)

17.12.19 Command Unlock Connection

Symbol: 

Function: This command releases a locked connection.

Call: Menu **CFC > Routing**, context menu > Routing

Requirements: A CFC editor is active. A connection or a connection mark is selected.

Changing the connections for automatic routing results in a locked connection. To perform automatic routing again, you must first release a locked connection.



You can also release this connection by clicking on the icon of a locked connection.

See also:

- PLC documentation: [Continuous Function Chart \(CFC\) \[▶ 113\]](#)
- PLC documentation: [CFC Editor \[▶ 666\]](#)
- PLC documentation: [CFC element Connection mark source/target \[▶ 679\]](#)

17.12.20 Command Show Next Collision

Function: The command indicates the next collision in the editor and marks the affected location.

Call:  button in the upper right corner of the editor

Requirements: A CFC editor is active, and there is at least one connection with collision.

This function is very useful if you work with large networks and only a subset is visible. A collision is also indicated by the symbol with a red frame in the upper right corner of the editor.

See also:

- PLC documentation: [Continuous Function Chart \(CFC\) \[▶ 113\]](#)
- PLC documentation: [CFC Editor \[▶ 666\]](#)

17.12.21 Command Select Connected Pins

Symbol: 

Function: The command selects all pins that are connected to the currently selected line or to the currently selected connection mark in the page-oriented CFC.


Call: Context menu

Requirements: A CFC editor or a page-oriented CFC editor is active. One line and thus exactly one connection or exactly one connection mark is selected.

See also:

- [Command Connect Selected Pins \[► 1017\]](#)
- PLC documentation: [Continuous Function Chart \(CFC\) \[► 113\]](#)
- PLC documentation: [CFC Editor \[► 666\]](#)

17.12.22 Command Use Attributed Component as Input

Symbol: 

Function: The command allows to connect a structure component to an input of scalar type.

Call: CFC > Pins menu, context menu > Pins

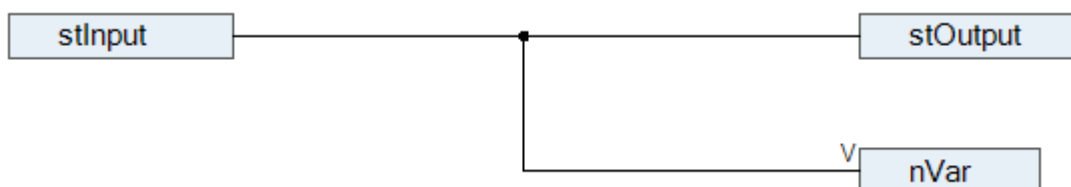
Requirements: A CFC editor is active and a function block input is selected.

The component of the structure that will be connected to the input of the subsequent function block must be provided with the {attribute 'ProcessValue'} attribute. The data type of the structure component must be compatible with the data type of the subsequent input. Inputs connected in this way are marked with the V symbol.

Sample

```
TYPE ST_Sample :
STRUCT
  {attribute 'ProcessValue'}
  nVar1 : INT;
  nVar2 : INT;
END_STRUCT
END_TYPE
```

```
PROGRAM MAIN
VAR
  stInput   : ST_Sample;
  stOutput  : ST_Sample;
  nVar      : INT;
END_VAR
```




If you do not execute the command **Use Attributed Component as Input** for this link, a compiler error is generated.

See also:

- PLC documentation: [Continuous Function Chart \(CFC\) \[► 113\]](#)
- PLC documentation: [CFC Editor \[► 666\]](#)

17.12.23 Command Reset Pins

Symbol: 

Function: The command restores deleted pins of a function block.

Call: Menu CFC > Pins, context menu > Pins


Requirements: A CFC editor is active and a function block is selected.

The command restores all inputs and outputs of the function block, as defined in its implementation.

See also:

- [Command Remove Unused Pins \[► 1020\]](#)
- PLC documentation: [Continuous Function Chart \(CFC\) \[► 113\]](#)
- PLC documentation: [CFC Editor \[► 666\]](#)

17.12.24 Command Remove Unused Pins

Symbol: 

Function: The command removes all unused pins of the selected element.


Call: Menu **CFC > Pins**, context menu > Pins

Requirements: A CFC editor is active. One element is selected.

See also:

- [Command Reset Pins \[► 1019\]](#)
- PLC documentation: [Continuous Function Chart \(CFC\) \[► 113\]](#)
- PLC documentation: [CFC Editor \[► 666\]](#)

17.12.25 Command Add Input Pin

Symbol: 

Function: The command adds a further input to the selected function block.


Call: Menu **CFC > Pins**, context menu > Pins

Requirements: A CFC editor is active. A function block is selected.

See also:

- [Command Add Output Pin \[► 1020\]](#)
- PLC documentation: [Continuous Function Chart \(CFC\) \[► 113\]](#)
- PLC documentation: [CFC Editor \[► 666\]](#)

17.12.26 Command Add Output Pin

Symbol: 

Function: The command adds a further output to the selected function block.


Call: Menu **CFC > Pins**, context menu > Pins

Requirements: A CFC editor is active. A suitable function block is selected.

See also:

- [Command Add Input Pin \[► 1020\]](#)
- PLC documentation: [Continuous Function Chart \(CFC\) \[► 113\]](#)
- PLC documentation: [CFC Editor \[► 666\]](#)

17.12.27 Command Route All Connections


Symbol: 

Function: The command undoes all manual changes to the connections in the program and restores the original state.

Call: Menu **CFC > Routing**, context menu > Routing

Requirements: A CFC editor is active.

TwinCAT cannot automatically route connections that are fixed with control points. You have to remove the control points before the command is executed. To do this, use the command Remove Control Point. In

addition, you have to disconnect manually changed connections, which are marked with the icon . To do this, use the command Unlock Connection.

See also:

- [Command Remove Control Point \[► 1021\]](#)
- [Command Unlock Connection \[► 1018\]](#)
- PLC documentation: [Continuous Function Chart \(CFC\) \[► 113\]](#)
- PLC documentation: [CFC Editor \[► 666\]](#)

17.12.28 Command Create Control Point

Symbol: 

Function: The command creates a control point on a connector.

Call: Context menu > Routing

Requirements: A CFC editor is active. The cursor is over a connection.

The control point is created at the point of the connection where the cursor is placed when the command is called. The command corresponds to the **Control point** element in the **Toolbox** window.

See also:

- [Command Remove Control Point \[► 1021\]](#)
- PLC documentation: [Continuous Function Chart \(CFC\) \[► 113\]](#)
- PLC documentation: [CFC Editor \[► 666\]](#)
- PLC documentation: [CFC element Control point \[► 676\]](#)

17.12.29 Command Remove Control Point

Function: The command removes a control point.

Call: Context menu > Routing

Requirements: A CFC editor is active. You have selected a connecting line.


If you move the mouse pointer over a selected connecting line, the existing control points are displayed with yellow circle symbols. Move the cursor to the control point to be deleted and execute the command from the context menu.

See also:

- PLC documentation: [Continuous Function Chart \(CFC\) \[► 113\]](#)
- PLC documentation: [CFC Editor \[► 666\]](#)

- PLC documentation: [CFC element Control point \[▶ 676\]](#)
- [Command Create Control Point \[▶ 1021\]](#)

17.12.30 Command Connection Mark

Symbol: 

Function: This command toggles the display of the connection between two elements between a connecting line and connection marks.

Call: **CFC** menu, context menu

Requirements: A CFC editor is active. A connection or a connection mark is selected.


If you have selected a connecting line, the command removes the line and adds a connection mark source to the output of one of the elements and a connection mark target to the input of the other. Both are assigned the same name "C-<n>" by default; n is a sequential number.

If you select a connection mark pair, the command converts these marks to a connecting line.

See also:

- PLC documentation: [Continuous Function Chart \(CFC\) \[▶ 113\]](#)
- PLC documentation: [CFC Editor \[▶ 666\]](#)
- PLC documentation: [CFC element Connection mark source/target \[▶ 679\]](#)

17.12.31 Command Create group

Symbol: 

Function: The command groups the selected elements.

Call: Menu **CFC > Group**, context menu > Group

Requirements: A CFC editor is active. Several elements are selected.

Grouped elements are moved together. The position of the elements is not influenced by the grouping.

See also:

- [Command Ungroup \[▶ 1022\]](#)
- PLC documentation: [Continuous Function Chart \(CFC\) \[▶ 113\]](#)
- PLC documentation: [CFC Editor \[▶ 666\]](#)

17.12.32 Command Ungroup

Symbol: 

Function: The command cancels a previously created grouping.

Call: Menu **CFC > Group**, context menu > Group

Requirements: A CFC editor is active. A grouping is selected.

See also:

- [Command Create group \[▶ 1022\]](#)
- PLC documentation: [Continuous Function Chart \(CFC\) \[▶ 113\]](#)

- PLC documentation: [CFC Editor](#) [▶ 666]

17.12.33 Command Edit Parameters

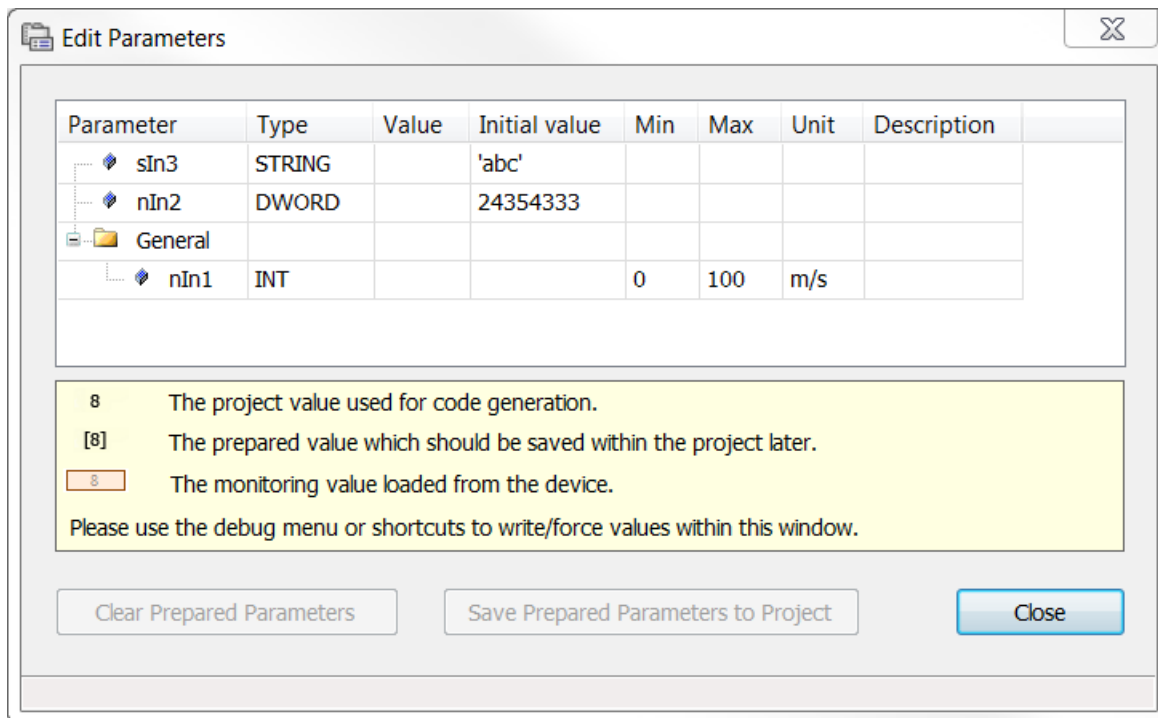
Function: The command opens the "Edit Parameters" dialog, in which you can change the constant input parameters of a function block.

Call: Menu **CFC > Edit parameters**, context menu > Edit parameters, click on function block field **Parameter**.

Requirements: A CFC editor is active. A function block is instantiated, which has VAR_INPUT CONSTANT variables in its declaration.

A function block with VAR_INPUT CONSTANT variables is indicated by TwinCAT by the word "Parameter" in the lower left corner of the function block.

Dialog Edit Parameters



Parameter	Name of the variable
Type	Data type of the variable
Value	Click in the field to enter a value.
Initial value	Initialization value
Category	Additional information about the parameters. These values are defined by attributes and cannot be changed in this dialog
Unit	
Min	
Max	
Delete prepared parameters	The command is active if you have written a prepared value (command Debug > Write values)

After exiting the field and exiting the dialog with **OK**, the value changes are applied to the project.

Example of a function block with constant inputs:

```
FUNCTION_BLOCK FB_Sample
VAR_INPUT CONSTANT
  {attribute 'parameterCategory':='General'}
```

```

{attribute 'parameterUnit':= 'm/s'}
{attribute 'parameterMinValue':= '0'}
{attribute 'parameterMaxValue':= '100'}
nIn1 : INT;
nIn2 : DWORD:=24354333;
sIn3 : STRING:='abc';
END_VAR

```



This functionality and the declaration of variables with the keyword VAR_INPUT CONSTANT only apply to the CFC editor. In the FBD editor, TwinCAT always shows all input parameters at the function block, irrespective of whether they are declared as VAR_INPUT or VAR_INPUT CONSTANT. Also, TwinCAT does not distinguish between text editors.

See also:

- [Command Save prepared parameters in the project \[► 1024\]](#)
- PLC documentation: [CFC editor in online mode \[► 673\]](#)

17.12.34 Command Force FB input



Available from TC3.1 Build 4026



This type of forcing uses a breakpoint internally and is therefore to be distinguished from forcing via the command **Force values**: Values that have been forced via the command **Force FB input** do not respond to the commands **Watch all forces** or **Cancel forcing for all values**.

Function: The command opens a dialog **Force value** for forcing the selected input of a function block. Forcing can be canceled with the same command and dialog.

Call: CFC menu, context menu

Requirements: The CFC editor is in online mode and the input of the function block is selected.

In the dialog **Force value** you can either enter a value with which the input of the function block is to be forced, or remove the currently forced value again.

After forcing, the input appears with a green background. Boolean inputs also get a small monitoring window with the forced value. The forced value is displayed in the column **Value** of monitoring views, i.e. in the declaration part of the program block or in a watch list.

Force value dialog

Expression	Name of the function block input
Type	Input data type

What do you want to do?

Set a new value to force	You can enter the desired new value in the input field. The format must correspond to the data type.
Remove value	Forcing at the input is canceled.

See also:

- PLC documentation: [Continuous Function Chart \(CFC\) \[► 113\]](#)
- PLC documentation: [CFC Editor \[► 666\]](#)

17.12.35 Command Save prepared parameters in the project

Function: The command applies the prepared parameter values in the project.

Call: CFC menu

Requirements: A CFC editor is active. Parameter values of function block instances were changed in online mode. The application is in offline mode.


If the values of constants on the controller differ from the values in the application, this is indicated by a red star to the right of the parameter field. Use the command **Apply prepared parameter values** to apply the control values to your application.

See also:

- [Command Edit Parameters \[► 1023\]](#)
- PLC documentation: [Changing constant input parameters of function block instances \[► 673\]](#)

17.13 FBD/LD/IL

17.13.1 Command Insert Contact (right)

Symbol: 

Function: The command inserts a contact to the right of the selected element.

Call: Menu **FBD/LD/IL**, context menu

Requirements: The LD editor is active. A line, a contact or a box is selected.

See also:

- PLC documentation: [FBD/LD/IL \[► 108\]](#)
- PLC documentation: [FBD/LD/IL editor \[► 651\]](#)
- PLC documentation: [LD element contact \[► 663\]](#)

17.13.2 Command Insert Network

Symbol: 

Function: The command inserts another network in the FBD/LD/IL editor.


Call: Menu **FBD/LD/IL**, context menu

Requirements: The FBD, LD or IL editor is active. No box is selected.

See also:

- PLC documentation: [FBD/LD/IL \[► 108\]](#)
- PLC documentation: [FBD/LD/IL editor \[► 651\]](#)
- PLC documentation: [FBD/LD/IL element network \[► 660\]](#)

17.13.3 Command Insert Network (below)

Symbol: 

Function: The command inserts another network in the FBD/LD/IL editor below the selected network.

Call: Menu **FBD/LD/IL**, context menu

Requirements: The FBD, LD or IL editor is active. A network is selected, but no box is selected.

See also:

- PLC documentation: [FBD/LD/IL \[► 108\]](#)

- PLC documentation: [FBD/LD/IL editor \[► 651\]](#)
- PLC documentation: [FBD/LD/IL element network \[► 660\]](#)

17.13.4 Command Toggle comment state

Symbol: 

Function: The command toggles comment state of the selected network.


Call: Menu **FBD/LD/IL**, context menu

Requirements: The FBD, LD or IL editor is active. A network is selected, but no box is selected.

See also:

- PLC documentation: [FBD/LD/IL \[► 108\]](#)
- PLC documentation: [FBD/LD/IL editor \[► 651\]](#)

17.13.5 Command Insert Assignment

Symbol: 

Function: The command inserts an assignment in the FBD or LD editor.

Call: Menu **FBD/LD/IL**, context menu

Requirements: The FBD, LD or IL editor is active. A network is selected, but no box is selected.



In IL, an assignment is programmed using the operators LD and ST.

See also:

- PLC documentation: [FBD/LD/IL \[► 108\]](#)
- PLC documentation: [FBD/LD/IL editor \[► 651\]](#)
- PLC documentation: [FBD/LD/IL element assignment \[► 661\]](#)

17.13.6 Command Insert Box

Symbol: 

Function: The command inserts a box that is available in the project at the end of the selected network.

Call: Menu **FBD/LD/IL**, context menu


Requirements: The FBD, LD or IL editor is active. A network is selected, but no box is selected.

When you select the command, the Input Assistant opens. There you can select the desired box.

See also:

- PLC documentation: [FBD/LD/IL \[► 108\]](#)
- PLC documentation: [FBD/LD/IL editor \[► 651\]](#)
- PLC documentation: [FBD/LD/IL element function block \[► 660\]](#)

17.13.7 Command Insert Box with EN/ENO

Symbol: 

Function: The command inserts a box with a Boolean input "Enable" and a Boolean output "Enable Out" at the end of the selected network.

Call: Menu **FBD/LD/IL**, context menu

Requirements: The FBD, LD or IL editor is active. A network is selected, but no box is selected.

See also:

- PLC documentation: [FBD/LD/IL \[▶ 108\]](#)
- PLC documentation: [FBD/LD/IL editor \[▶ 651\]](#)
- PLC documentation: [FBD/LD/IL element function block with EN/ENO \[▶ 661\]](#)

17.13.8 Command Insert Empty Box

Symbol: 

Function: The command inserts an empty box at the end of the currently selected network.


Call: Menu **FBD/LD/IL**, context menu

Requirements: The FBD, LD or IL editor is active. A network is selected, but no box is selected.

See also:

- PLC documentation: [FBD/LD/IL \[▶ 108\]](#)
- PLC documentation: [FBD/LD/IL editor \[▶ 651\]](#)
- PLC documentation: [FBD/LD/IL element function block \[▶ 660\]](#)

17.13.9 Command Insert Box with EN/ENO

Symbol: 

Function: The command inserts an empty box with a Boolean input "Enable" and a Boolean output "Enable Out" at the end of the selected network.

Call: Menu **FBD/LD/IL**, context menu

Requirements: An FBD editor, LD editor or IL editor is active. A network must be selected. No other box may be selected.

If "Enable" has the value FALSE when the box is called, the operations defined in the box are not executed. Otherwise, i.e. if "Enable" is TRUE, these operations are executed. The ENO output acts as a repeater of the EN input.

See also:

- PLC documentation: [FBD/LD/IL \[▶ 108\]](#)
- PLC documentation: [FBD/LD/IL editor \[▶ 651\]](#)
- PLC documentation: [FBD/LD/IL element function block with EN/ENO \[▶ 661\]](#)

17.13.10 Command Insert jump

Symbol: 

Function: The command inserts a jump element before the currently selected element.


Call: Menu **FBD/LD/IL**, context menu

Requirements: The FBD, LD or IL editor is active. A connector is selected.

See also:

- PLC documentation: [FBD/LD/IL \[► 108\]](#)
- PLC documentation: [FBD/LD/IL editor \[► 651\]](#)
- PLC documentation: [FBD/LD/IL element jump \[► 662\]](#)

17.13.11 Command Insert label

Symbol: 

Function: The command inserts a label in the currently selected network.


Call: Menu **FBD/LD/IL**, context menu

Requirements: The FBD, LD or IL editor is active. A network is selected. No label is selected.

See also:

- PLC documentation: [FBD/LD/IL \[► 108\]](#)
- PLC documentation: [FBD/LD/IL editor \[► 651\]](#)
- PLC documentation: [FBD/LD/IL element label \[► 661\]](#)

17.13.12 Command Insert Return

Symbol: 

Function: The command inserts a "Return" element at the selected position.


Call: Menu **FBD/LD/IL**, context menu

Requirements: The FBD, LD or IL editor is active. A box output is selected.

See also:

- PLC documentation: [FBD/LD/IL \[► 108\]](#)
- PLC documentation: [FBD/LD/IL editor \[► 651\]](#)
- PLC documentation: [FBD/LD/IL element return \[► 662\]](#)

17.13.13 Command Insert Input

Symbol: 

Function: The command adds an additional input to an expandable box (ADD, OR, AND, MUL, SEL) above the selected input.

Call: Menu **FBD/LD/IL**

Requirements: The FBD/LD editor is active. A box input is selected.


If a box is selected, the command **Append name input** is available in the context menu. The input is inserted at the bottom of the box.

See also:

- PLC documentation: [FBD/LD/IL \[► 108\]](#)

- PLC documentation: [FBD/LD/IL editor \[► 651\]](#)

17.13.14 Command Insert box in parallel (below)

Symbol: 

Function: The command inserts an empty box in parallel to the selected box.

Call: Menu **FBD/IL/LD**, context menu

Requirements: A box is selected in the LD editor.

See also:

- PLC documentation: [FBD/LD/IL Editor \[► 651\]](#)

17.13.15 Command Insert Coil

Symbol: 

Function: The command inserts a coil in the network.

Call: Menu **FBD/LD/IL**, context menu

Requirements: The LD editor is active. A network, a coil or a connector is selected, but no box is selected.

See also:

- PLC documentation: [FBD/LD/IL \[► 108\]](#)
- PLC documentation: [FBD/LD/IL editor \[► 651\]](#)
- PLC documentation: [LD element coil \[► 664\]](#)

17.13.16 Command Insert Set coil

Symbol: 

Function: The command inserts a set coil in the network.

Call: Menu **FBD/LD/IL**, context menu

Requirements: The LD editor is active. A network, a coil or a line is selected, but no box is selected.

See also:

- PLC documentation: [FBD/LD/IL \[► 108\]](#)
- PLC documentation: [FBD/LD/IL editor \[► 651\]](#)
- PLC documentation: [LD element coil \[► 664\]](#)

17.13.17 Command Insert Reset coil

Symbol: 

Function: The command inserts a reset coil in the network.

Call: Menu **FBD/LD/IL**, context menu

Requirements: The LD editor is active. A network, a coil or a line is selected, but no box is selected.

See also:

- PLC documentation: [FBD/LD/IL \[▶ 108\]](#)
- PLC documentation: [FBD/LD/IL editor \[▶ 651\]](#)
- PLC documentation: [LD element coil \[▶ 664\]](#)

17.13.18 Command Insert Contact

Symbol: 

Function: The command inserts a contact to the left of the selected element.


Call: Menu **FBD/LD/IL**, context menu

Requirements: The LD editor is active. A line or a contact is selected.

See also:

- PLC documentation: [FBD/LD/IL \[▶ 108\]](#)
- PLC documentation: [FBD/LD/IL editor \[▶ 651\]](#)
- PLC documentation: [LD element contact \[▶ 663\]](#)

17.13.19 Command Insert Contact Parallel (below)

Symbol: 

Function: The command inserts a contact with lines in parallel below the selected element.

Call: Menu **FBD/LD/IL**, context menu

Requirements: The LD editor is active. A line, a contact or a box is selected.

i You can program closed parallel branches in an LD network as a Short Circuit Evaluation (SCE) or an OR construct. SCE branches are represented by double vertical lines, OR branches with single lines. See the help page for "[Closed line branches \[▶ 665\]](#)".

See also:

- PLC documentation: [FBD/LD/IL \[▶ 108\]](#)
- PLC documentation: [FBD/LD/IL editor \[▶ 651\]](#)
- PLC documentation: [LD element contact \[▶ 663\]](#)

17.13.20 Command Insert Contact Parallel (above)

Symbol: 

Function: The command inserts a contact with lines in parallel above the selected element.

Call: Menu **FBD/LD/IL**, context menu


Requirements: The LD editor is active. A line, a contact or a box is selected.

i You can program closed parallel branches in an LD network as a Short Circuit Evaluation (SCE) or an OR construct. SCE branches are represented by double vertical lines, OR branches with single lines. See the help page for "[Closed line branches \[▶ 665\]](#)".

See also:

- PLC documentation: [FBD/LD/IL \[► 108\]](#)
- PLC documentation: [FBD/LD/IL editor \[► 651\]](#)
- PLC documentation: [LD element contact \[► 663\]](#)

17.13.21 Command Insert Negated Contact

Symbol: 

Function: The command inserts a negated contact to the left of the selected element.


Call: Menu **FBD/LD/IL**, context menu

Requirements: The LD editor is active. A line or a contact is selected.

See also:

- PLC documentation: [FBD/LD/IL \[► 108\]](#)
- PLC documentation: [FBD/LD/IL editor \[► 651\]](#)
- PLC documentation: [LD element contact \[► 663\]](#)

17.13.22 Command Insert Negated Contact Parallel (below)

Symbol: 

Function: The command inserts a negated contact with lines in parallel below the selected element.

Call: Menu **FBD/LD/IL**, context menu

Requirements: The LD editor is active. A line, a contact or a box is selected.

See also:

- PLC documentation: [FBD/LD/IL \[► 108\]](#)
- PLC documentation: [FBD/LD/IL editor \[► 651\]](#)
- PLC documentation: [LD element contact \[► 663\]](#)

17.13.23 Command Paste Contacts: Paste below

Function: The command pastes a previously copied contact with lines below the selected element.

Call: Menu **FBD/LD/IL**, context menu

Requirements: The LD editor is active.

See also:

- PLC documentation: [FBD/LD/IL \[► 108\]](#)
- PLC documentation: [FBD/LD/IL editor \[► 651\]](#)
- PLC documentation: [LD element contact \[► 663\]](#)

17.13.24 Command Paste Contacts: Paste above

Function: The command pastes a previously copied contact with lines above the selected element.

Call: Menu **FBD/LD/IL**, context menu

Requirements: The LD editor is active. A line or a contact is selected.

See also:

- PLC documentation: [FBD/LD/IL](#) [▶ 108]
- PLC documentation: [FBD/LD/IL editor](#) [▶ 651]
- PLC documentation: [LD element contact](#) [▶ 663]

17.13.25 Command Paste Contacts: Paste right (after)

Function: The command pastes a previously copied contact to the right of the selected element.

Call: Menu **FBD/LD/IL**, context menu

Requirements: The LD editor is active. A line or a contact is selected.

See also:

- PLC documentation: [FBD/LD/IL](#) [▶ 108]
- PLC documentation: [FBD/LD/IL editor](#) [▶ 651]
- PLC documentation: [LD element contact](#) [▶ 663]

17.13.26 Command Insert IL line below

Symbol: 

Function: The command inserts an instruction line below the selected line.

Call: Menu **FBD/LD/IL**, context menu

Requirements: The IL editor is active. A line is selected.

See also:

- PLC documentation: [FBD/LD/IL](#) [▶ 108]
- PLC documentation: [FBD/LD/IL editor](#) [▶ 651]

17.13.27 Command Delete IL line

Symbol: 

Function: The command deletes the selected instruction line.


Call: Menu **FBD/LD/IL**, context menu

Requirements: The IL editor is active. A line is selected.

See also:

- PLC documentation: [FBD/LD/IL](#) [▶ 108]
- PLC documentation: [FBD/LD/IL editor](#) [▶ 651]

17.13.28 Command Negation

Symbol: 

Function: The command negates the following elements:

- Input/output of a box
- Jump

- Return
- Coil

Call: Menu **FBD/LD/IL**, context menu

Requirements: The FBD or LD editor is active. The corresponding element is selected.

See also:

- PLC documentation: [FBD/LD/IL \[► 108\]](#)
- PLC documentation: [FBD/LD/IL editor \[► 651\]](#)

17.13.29 Command Edge detection

Symbol FBD: 

Symbol LD: 

Function: The command inserts an edge detection before the selected box input or box output. A distinction can be made between the following functionalities by executing the command once, twice or three times:

- The edge detection inserted when the command is executed once serves to detect a rising edge. The arrow of the symbol points to the right.
- If the command is executed again, the edge detection is reversed, so that falling edges are detected. The arrow of the symbol points to the left.
- If the command is executed one more time, the edge detection and the symbol are removed.

Call: Menu **FBD/LD/IL**, context menu

Requirements: The FBD or LD editor is active. A box input or output is selected.

See also:

- PLC documentation: [FBD/LD/IL \[► 108\]](#)
- PLC documentation: [FBD/LD/IL editor \[► 651\]](#)

17.13.30 Command Set/Reset

Symbol: 

Function: For an element with a Boolean output, the command toggles between reset, set and no label.


Call: Menu **FBD/LD/IL**, context menu

Requirements: The FBD or LD editor is active. An element with Boolean output is selected.

See also:

- PLC documentation: [FBD/LD/IL \[► 108\]](#)
- PLC documentation: [FBD/LD/IL editor \[► 651\]](#)

17.13.31 Command Set output connection

Symbol: 

Function: The command assigns the selected box output as interconnecting box output.


Call: Menu **FBD/LD/IL**, context menu

Requirements: The FBD or LD editor is active. One of several box outputs is selected.

See also:

- PLC documentation: [FBD/LD/IL](#) [► 108]
- PLC documentation: [FBD/LD/IL editor](#) [► 651]

17.13.32 Command Insert Branch

Symbol: 

Function: The command creates an open branch on the selected line.

Call: Menu **FBD/LD/IL**, context menu

Requirements: The FBD or LD editor is active. A box input or output is selected.

See also:

- PLC documentation: [FBD/LD/IL](#) [► 108]
- PLC documentation: [FBD/LD/IL editor](#) [► 651]
- PLC documentation: [FBD/LD/IL element line branch](#) [► 662]

17.13.33 Command Insert Branch above

Symbol: 

Function: The command creates a branch above the selected open branch.


Call: Menu **FBD/LD/IL**, context menu

Requirements: The FBD or LD editor is active. An open branch is selected.

See also:

- PLC documentation: [FBD/LD/IL](#) [► 108]
- PLC documentation: [FBD/LD/IL editor](#) [► 651]
- PLC documentation: [FBD/LD/IL element line branch](#) [► 662]

17.13.34 Command Insert Branch below

Symbol: 

Function: The command creates a branch below the selected open branch.


Call: Menu **FBD/LD/IL**, context menu

Requirements: The FBD or LD editor is active. An open branch is selected.

See also:

- PLC documentation: [FBD/LD/IL](#) [► 108]
- PLC documentation: [FBD/LD/IL editor](#) [► 651]
- PLC documentation: [FBD/LD/IL element line branch](#) [► 662]

17.13.35 Command Set Branch Start Point

Symbol: 

Function: The command sets the start point of a branch on the selected line.


Call: Menu **FBD/LD/IL**, context menu

Requirements: The LD editor is active. A line is selected.

See also:

- PLC documentation: [FBD/LD/IL \[► 108\]](#)
- PLC documentation: [FBD/LD/IL editor \[► 651\]](#)
- PLC documentation: [Closed line branch \[► 665\]](#)

17.13.36 Command Set Branch End Point

Symbol: 

Function: The command sets the end point of a branch on the selected line.

Call: Menu **FBD/LD/IL**, context menu

Requirements: The LD editor is active. A line is selected. A start point for the branch was set.

See also:

- PLC documentation: [FBD/LD/IL \[► 108\]](#)
- PLC documentation: [FBD/LD/IL editor \[► 651\]](#)
- PLC documentation: [Closed line branch \[► 665\]](#)

17.13.37 Command Toggle parallel mode

Function: This command toggles a parallel branch between an OR construct and the Short Circuit Evaluation (SCE).

Call: **FBD/LD/IL** menu, context menu

Requirements: The FBD/LD/IL editor is active. A vertical line of a parallel branch is selected.



You can program closed parallel branches in an LD network as a Short Circuit Evaluation (SCE) or an OR construct. SCE junctions are represented by double vertical lines, OR junctions with single lines. See the help page for "[Closed branches \[► 665\]](#)".

See also:

- PLC documentation: [FBD/LD/IL \[► 108\]](#)
- PLC documentation: [FBD/LD/IL editor \[► 651\]](#)
- PLC documentation: [Closed branch \[► 665\]](#)

17.13.38 Command Update parameters

Function: The command applies changes in the declaration of the selected element to the graphic.

Call: Menu **FBD/LD/IL**, context menu


Requirements: The FBD, LD or CFC editor is active. A box is selected. An extended change to the declaration was carried out.

The command checks whether a box and its declaration match in the declaration editor. The change is only applied to the box if the declaration was extended. Deletions and overwrites are not updated.

See also:

- PLC documentation: [FBD/LD/IL \[► 108\]](#)
- PLC documentation: [FBD/LD/IL editor \[► 651\]](#)

17.13.39 Command Remove unused FB call parameters

Symbol: 

Function: The command deletes inputs and outputs of the selected function block, to which no variable and no value was assigned. However, the default inputs and outputs of the function block are always retained.


Call: Menu **FBD/LD/IL**, context menu

Requirements: The FBD or LD editor is active. A function block is selected. The function block has interfaces that are not assigned a value.

See also:

- PLC documentation: [FBD/LD/IL \[► 108\]](#)
- PLC documentation: [FBD/LD/IL editor \[► 651\]](#)

17.13.40 Command Repair POU

Symbol: 

Function: The command repairs internal inconsistencies on the selected function block.

Call: Menu **FBD/LD/IL**, context menu

Requirements: The FBD or LD editor is active. The faulty function block is selected. The editor has detected internal inconsistencies in the programming block, which may be resolved automatically. TwinCAT reports the inconsistencies in the **Error list** view.

This situation is conceivable when you edit a project created with an older version of the programming system that has not yet treated the inconsistency as an error.

See also:

- PLC documentation: [FBD/LD/IL \[► 108\]](#)
- PLC documentation: [FBD/LD/IL editor \[► 651\]](#)

17.13.41 Command View as function block diagram

NOTICE

Loss of data

An error-free conversion requires syntactically correct code. Otherwise, parts of the implementation may be lost.

Function: The command converts the active Instruction List or the active Ladder Diagram to the function block diagram.

Call: Menu **FBD/LD/IL > View**

Requirements: The LD or IL editor is active.

See also:

- PLC documentation: [FBD/LD/IL \[► 108\]](#)
- PLC documentation: [FBD/LD/IL editor \[► 651\]](#)

17.13.42 Command View as ladder logic

NOTICE

Loss of data

An error-free conversion requires syntactically correct code. Otherwise, parts of the implementation may be lost.

Function: The command converts the current function block code or the active Instruction List to a Ladder Diagram.

Call: Menu **FBD/LD/IL > View**

Requirements: The FBD or IL editor is active.

See also:

- PLC documentation: [FBD/LD/IL \[► 108\]](#)
- PLC documentation: [FBD/LD/IL editor \[► 651\]](#)

17.13.43 Command View as instruction list



If required, IL can be enabled via the TwinCAT options.

NOTICE

Loss of data

An error-free conversion requires syntactically correct code. Otherwise, parts of the implementation may be lost.

Function: The command converts the active function block code or the active Ladder Diagram to an Instruction List.

Call: Menu **FBD/LD/IL > View**

Requirements: The FBD or LD editor is active.

See also:

- PLC documentation: [FBD/LD/IL \[► 108\]](#)
- PLC documentation: [FBD/LD/IL editor \[► 651\]](#)

17.13.44 Command Go To

Symbol:

Function: The command lets you jump to any network.

Call: Menu **FBD/LD/IL**

Requirements: The LD, FBD editor or IL editor is active. A network is selected.

The command opens a dialog with an input field. Enter the number of the desired network in the input field.

See also:

- PLC documentation: [FBD/LD/IL \[► 108\]](#)
- PLC documentation: [FBD/LD/IL editor \[► 651\]](#)

17.14 Ladder editor

The commands for working in the Ladder editor are available in the **Ladder** menu and in the corresponding context menus.

17.14.1 Command Outcommented

Symbol: 

Function: The command toggles a network [**▶ 1040**] between the commented out and not commented out states.

Call: **Ladder** menu; context menu

The network contents are displayed in gray and the texts in italics. The network is not taken into consideration during processing.

17.14.2 Command Negate

Symbol: 

Function: The command negates the following elements:

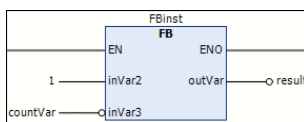
- Input/output of a box
- Jump
- Return
- Coil
- Contact
- Variable

Call: **Ladder** menu; context menu

Requirements: The element to be negated is selected.

Examples:

Negated input and negated output on one function block:



Negated contact:



Negated coil:



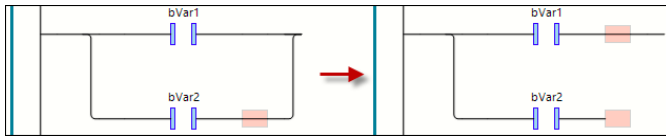
17.14.3 Command Open Parallel Branch

Symbol: 

Function: The command opens a closed parallel branch.

Call: **Ladder** menu; context menu

Requirements: One of the two lines of the branch to be reopened has to be selected.



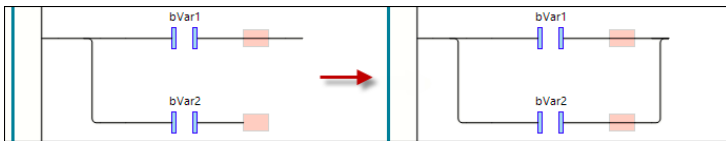
17.14.4 Command Close Parallel Branch

Symbol: 

Function: The command closes the open parallel branch.

Call: Ladder menu; context menu

Requirements: Both lines of the branch to be closed have to be selected:



Alternatively, an open branch could also be closed by dragging the selection marker of one branch to the selection marker of the other branch.

17.14.5 Command Set/Reset – Set, Set/Reset – Reset

Symbol: Set , Reset 

Function: The Set command provides the element selected in the editor with a Set modifier. The Reset command provides the element with a Reset modifier.

You can use it to make a "set coil" from a [coil](#) [[▶ 1041](#)] and a non-primary box, for example. The command is available only at the appropriate positions.

Call: Ladder menu, Set/Reset submenu; context menu

17.14.6 Command Edge Detection – Rising Edge

Symbol: 

Function: The command inserts a detection for a rising edge before the selected box input or contact.

Call: Ladder menu; context menu

Requirements: A box input or contact is selected.

17.14.7 Command Edge Detection – Falling Edge

Symbol: 

Function: The command inserts a detection for a falling edge before the selected box input or contact.

Call: Ladder menu; context menu

Requirements: A box input or contact is selected.

17.14.8 Command EN/ENO: EN

Symbol: 

Function: The command is used to add or remove a Boolean input "Enable" at the selected box.

Call: Ladder menu, EN/ENO submenu; context menu

The Boolean EN input controls the execution of the box. If the EN value is FALSE during the box call, then the operations which are defined in the box are not executed. These operations are executed if EN is TRUE.

If a box has an EN input, a ENO output [[▶_1040](#)] can also be added to it: ENO gets the same value as EN.

17.14.9 Command EN/ENO: ENO

Symbol: 

Function: The command adds an ENO output to a box with EN input.


Call: Ladder menu, EN/ENO submenu; context menu

Requirements: A box with an EN input is selected.

The ENO output has the same value as the EN input.

17.14.10 Command Insert Network

Symbol: 

Function: The command inserts an additional network in the Ladder editor. You recognize the possible insertion positions by the plus symbol at the cursor  when you drag the element over the implementation part.

Call: Ladder menu; context menu

Requirements: No box is selected.

17.14.10.1 Element: Network

A network is the basic unit of an LD program. In the Ladder editor, the networks are arranged in a list below each other. Each network is provided with an incremented network number on the left side and can include the following: logical and arithmetic expressions, program/function/function block calls, jumps or return statements.

You can provide each network with a title, comment or jump label.

Network title: Double-click the first line at the top of a network to specify it.


Network comment: Double-click the second line at the top of a network to specify it.

Jump label: Double-click the third line at the top of a network to specify it. The jump label can then be specified as a target at the Jump [[▶_1042](#)] element.

In the TwinCAT options, category Ladder [[▶_983](#)] you define whether network title, network comment and separator lines between the individual networks are displayed in the editor.

17.14.11 Command Insert Contact

Symbol:  , in the editor 

Function: The command inserts a contact in the network, to the left of the selected element. When you drag the element from the toolbox to the implementation part, the plus symbol at the cursor  helps you recognize the possible insertion positions.


Call: Ladder menu; context menu

Requirements: A line or a contact is selected.

17.14.11.1 Element: Contact

A contact passes on the signal TRUE (ON) or FALSE (OFF) from left to right until the signal reaches a coil in the right side of the network. To this end, the contact is assigned a Boolean variable that contains the signal. To do this, replace the ??? placeholder above the contact with the name of a Boolean variable.


You can arrange several contacts in series or in parallel. If two parallel contacts are used, only one of them has to have the value TRUE for ON to be passed to the right. If contacts are connected in series, all contacts must be set to TRUE for ON to be passed to the right from the last contact of the series. You can therefore use LD to program electrical parallel and series circuits.

A negated contact  passes the TRUE signal when the variable value is FALSE. You can use the **Ladder** → **Negate** command to enable or disable negation of an inserted contact.

17.14.12 Command Insert Coil

Symbol: 

Symbol in the editor: 



Function: The command inserts a coil in the network. When you drag the element from the toolbox to the implementation part, the plus symbol at the cursor  helps you recognize the possible insertion positions.

Call: **Ladder** menu; context menu

Requirements: A network, a coil or a connector is selected. No box is selected.

17.14.12.1 Element: Coil

A coil takes the value delivered from the left and stores it in the Boolean variable assigned to it. Its input can have the value TRUE (ON) or FALSE (OFF).

In a negated  coil  the negated value of the incoming signal is stored in the Boolean variable assigned to the coil.


With the commands of the Set/Reset  submenu you can define a "set coil" or "reset coil":

Set coil: If the value TRUE arrives at a set coil, the coil retains the value TRUE. As long as the application is running, the value can no longer be overwritten here.



Reset coil: If the value TRUE arrives at a reset coil, the coil retains the value FALSE. As long as the application is running, the value can no longer be overwritten here.

17.14.13 Command Insert Box

Symbol: 

Function: The command inserts a box, which is available at the end of the selected network. When you drag the element from the toolbox to the implementation part, the plus symbol at the cursor  helps you recognize the possible insertion positions.

Call: **Ladder** menu; context menu

The command inserts an EN/ENO box by default. The **Ladder** EN/ENO menu commands can be used to remove and add the EN  and ENO  modifiers.

NOT CURRENTLY IMPLEMENTED: If it is a box defined in the project, the usage in the Ladder must be updated after a change to this box.


17.14.13.1 Element: Box

A box element can represent additional functions: IEC function blocks, IEC functions, library function blocks or operators, for example.

If the box also provides an image file, the box icon can be displayed inside the box. Prerequisite: The option **Show box icon** is enabled in the TwinCAT options, category **Ladder**.

17.14.14 Command Insert Jump

Symbol: →

Function: The command inserts a jump element before the selected element. When you drag the element from the toolbox to the implementation part, the plus symbol at the cursor  helps you recognize the possible insertion positions.

Call: **Ladder** menu; context menu

Requirements: A connector is selected.


17.14.14.1 Element: Jump

In LD, a jump is inserted either directly before an input, directly after an output or at the end of the network, depending on the current cursor position.

Directly after the jump element you enter the jump label of a network as jump destination.

17.14.15 Command Insert Return

Symbol: 

Function: The command inserts a **Return** element in the selected location. When you drag the element from the toolbox to the implementation part, the plus symbol at the cursor  helps you recognize the possible insertion positions.


Call: **Ladder** menu; context menu

17.14.15.1 Element: Return

The element immediately interrupts the execution of the Ladder box when the input of the **Return** element becomes TRUE. You can place the **Return** statement parallel to or after the preceding elements.

17.14.16 Command Insert Input

Symbol: 

Function: The command inserts an input at the selected position. When you drag the element from the toolbox to the implementation part, the plus symbol at the cursor  helps you recognize the possible insertion positions. You can replace the initially displayed question marks ??? with a variable name or a constant.


Call: **Ladder** menu; context menu

17.14.16.1 Element: Input

The element is used to assign a variable value to another element in the signal flow. It has a continuing line to the right.

17.14.17 Command Insert Output

Symbol: 

Function: The command inserts an output at the selected position. When you drag the element from the toolbox to the implementation part, the plus symbol at the cursor  helps you recognize the possible insertion positions. You can replace the initially displayed question marks ??? with a variable name or a constant.

Call: Ladder menu; context menu

17.14.17.1 Element: Output

The element is used to assign a signal coming from the left to the variable associated with the element. It has an input line coming from the left.

17.14.18 Command Convert to New Ladder

Available in the FBD/LD editor.

17.15 Declarations

17.15.1 Command Paste

Symbol: 

Function: The command inserts a new line for a variable declaration in the tabular declaration editor and the input field for the variable name opens.

Call: Button in the declaration header of the tabular declaration editor, context menu in the tabular declaration editor

To edit the other fields of the declaration line, double-click the fields and select the information from the selection lists or using the corresponding dialogs.

See also:

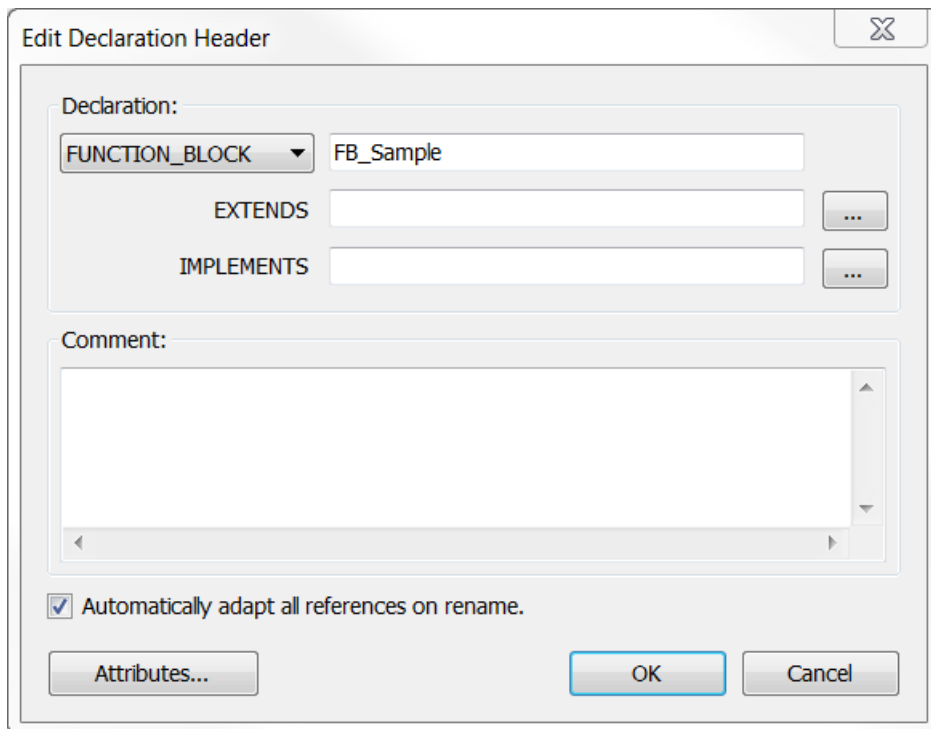
- PLC documentation: [Using the declaration editor \[► 70\]](#)

17.15.2 Command Edit the declaration header

Function: The command opens the dialog **Edit the declaration header**, which is used in the tabular declaration editor to configure the header of a POU.

Call: Mouse click on the header bar of the tabular declaration editor, context menu in the tabular declaration editor

Dialog Edit the declaration header



Declaration	<p>Selection list for changing the POU type</p> <ul style="list-style-type: none"> • PROGRAM • FUNCTION_BLOCK <ul style="list-style-type: none"> ◦ EXTENDS: Input field for a basic function block ◦ IMPLEMENTS: Input field for an interface • FUNCTION <ul style="list-style-type: none"> ◦ Return type <p>Input field with current POU name: You can change the name of the POU.</p>
Automatically adjust all references when renaming	<p><input checked="" type="checkbox"/> : The Refactoring dialog opens.</p> <p><input type="checkbox"/> : The renaming takes effect only in the declaration header of the POU.</p>
Attributes	The Attributes dialog opens for entering attributes and pragmas.

See also:

- PLC documentation: [Using the Declaration Editor \[▶ 70\]](#)
- PLC documentation: [Pragmas \[▶ 793\]](#)
- PLC documentation: [Refactoring \[▶ 160\]](#)

17.15.3 Command Move Down

Symbol: 

Function: The command moves a variable declaration down one line.


Call: Button in the declaration header of the tabular declaration editor, context menu in the tabular declaration editor

Requirement: A line with a variable declaration is selected in the tabular declaration editor.

See also:

- PLC documentation: [Using the declaration editor \[► 70\]](#)

17.15.4 Command Move Up

Symbol: 

Function: The command moves a variable declaration up one line.

Call: Button in the declaration header of the tabular declaration editor, context menu in the tabular declaration editor


Requirement: A line with a variable declaration is selected in the tabular declaration editor.

See also:

- PLC documentation: [Using the declaration editor \[► 70\]](#)

17.16 Textlist

17.16.1 Command Add Language

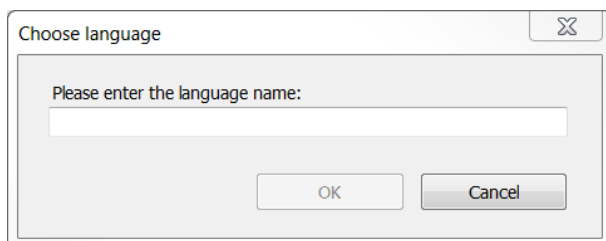
Symbol: 

Function: This command adds an additional language column in the text list.

Call: Menu **Textlist**, context menu

Requirement: A text list or a global text list is open and active.


Enter an abbreviation for the new language in the **Choose language** dialog, for example "en-US". TwinCAT inserts the abbreviation as a column heading.



See also:

- PLC documentation: [Managing text in a text list \[► 138\]](#)

17.16.2 Command Remove Language

Symbol: 

Function: The command removes the selected language column from the text list.


Call: Menu **Textlist**, context menu

Requirement: A text list or a global text list is open and active. A field in the column of the language that you want to remove is selected.

See also:

- PLC documentation: [Managing text in a text list \[► 138\]](#)

17.16.3 Command Insert Text

Symbol: 

Function: The command inserts a new line above the selected line in the text list. An input field opens under Standard, in which you enter the source text.

Call: Menu **Textlist**

Requirement: A text list (no GlobalTextList) is open and active. A field is selected in the table.

See also:

- PLC documentation: [Managing text in a text list \[▶ 138\]](#)

17.16.4 Command Import/Export Text Lists

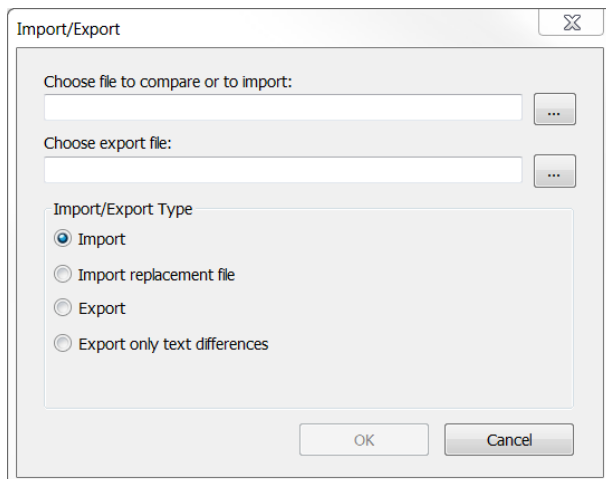
Symbol: 



Function: The command exports an active text list, imports a file or synchronizes a text list with a file. The file is in CSV format. The **Import/Export** dialog offers options for this purpose.

Call: Menu **Textlist**, context menu

Requirement: A text list or a global text list is open and active.

Import/Export dialog



Choose file to compare or to import	File, which TwinCAT reads.  opens the Choose textlist file dialog, in which you can select a file.
Choose export file	File to which TwinCAT writes.  opens the Choose textlist file dialog, in which you can select a file and a directory.

Import/Export Type:

Import	<p>Requirement: A file is selected in Choose file to compare or to import.</p> <p>The file may contain text list records for the global text list of for text lists.</p> <p>Global text list:</p> <ul style="list-style-type: none"> • TwinCAT reads the file, compares the text list records for the same source text and applies differences in the translations. TwinCAT overwrites the translations in the project, if necessary. <p>Text lists:</p> <ul style="list-style-type: none"> • TwinCAT reads the file, compares the text list records for the same IDs and applies differences in the source text and the translation to the project. TwinCAT overwrites the text list records in the project, if necessary. • If the file contains a new ID, the text list entry is imported into the text list of the project, and the text list is amended.
Import replacement file	<p>Requirement: A replacement file is selected in Choose file to compare or to import.</p> <p>The replacement file contains replacements for the global text list.</p> <p>TwinCAT processes the replacement file line-by-line and implements the specified replacements in the global text list. The structure of the replacement file is described in section Managing Static Text in Global Text Lists [► 142].</p>
Export	<p>Requirement: The file to which TwinCAT writes is selected in "Choose export file".</p> <p>TwinCAT exports all texts from all text lists of the current project. All languages available in the project are inserted as columns in the export file. The file can be used to translate the language-dependent text externally.</p>
Export only text differences	<p>Requirement: An import file for the comparison is selected in Choose file to compare or to import. An export file to which TwinCAT writes is selected in Choose export file.</p> <p>TwinCAT reads the import file and compares the lines of the active text list with it. TwinCAT ignores any matching lines. If lines differ, TwinCAT writes the line to the export file and applies translations from the text list. TwinCAT applies the translations from the import file and overwrites them, if applicable.</p>

See also:

- PLC documentation: [Managing text in a text list \[► 138\]](#)

17.16.5 Command Remove Unused Text List Records

Symbol: 

Function: The command checks whether a text list record in the project will be used as static text. If not, TwinCAT removes it from the text list.

Call: Menu **Textlist**, context menu

Requirement: The global text list is open and active. A field is selected in the table.

See also:

- PLC documentation: [Managing text in a text list \[► 138\]](#)

17.16.6 Command Check Visualization Text Ids

Symbol: 

Function: The command checks whether the ID of a text list record in the project is correct and reports the result.

Call: Menu **Textlist**, context menu


Requirement: The global text list is open and active. A field is selected in the table.

If TwinCAT detects that the global text list and the static texts of the visualizations do not match, this may be because the global text list is or was read-only. A requirement is that you have set up user management in the project.

See also:

- PLC documentation: [Managing text in a text list \[► 138\]](#)

17.16.7 Command Update Visualization Text Ids

Symbol: 

Function: The command updates all inconsistent IDs in a static text list.

Call: Menu **Textlist**, context menu


Requirement: The global text list is open and active. A field is selected in the table. The object is read-only.

If TwinCAT detects that the global text list and the static texts of the visualizations do not match, this may be because the global text list is or was read-only. A requirement is that you have set up user management in the project.

See also:

- PLC documentation: [Managing text in a text list \[► 138\]](#)

17.16.8 Command Export All

Symbol: 

Function: The command exports all text lists of the project.

Call: Menu **Textlist**, context menu

Requirement:

- A text list or a global text list is open and active.
- The visualization does not encode the text characters in Unicode.

For each text list TwinCAT creates a file as plain text in .txt format. The name of text list becomes the name of the file. The file is saved in the directory of the TwinCAT project.

A controller can read and use this format. You can copy the file to a controller, for example, and configure it via a setting in the Visualization Manager, such that the text lists are not transferred again when the PLC project is loaded.

See also:

- PLC documentation: [Managing text in a text list \[► 138\]](#)

17.16.9 Command Export All Unicode

Symbol: 

Function: The command exports all text lists of the project.

Call: Menu **Textlist**, context menu

Requirement:

- A text list or a global text list is open and active.

- The visualization encodes the text characters in Unicode.
 - The option **Use Unicode strings** is enabled in the Visualization Manager.

For each text list TwinCAT creates a file as plain text in .txt format. The name of text list becomes the name of the file. The file is saved in the directory of the TwinCAT project.

A controller can read and use this format. You can copy the file to a controller, for example, and configure it via a setting in the Visualization Manager, such that the text lists are not transferred again when the PLC project is loaded.

See also:

- PLC documentation: [Managing text in a text list \[► 138\]](#)

17.16.10 Command Add text list support

Symbol: 

Function: The command adds text list support to the selected DUT object of type **Enumeration**.

Call: Context menu of a standard DUT object of type **Enumeration** ()

Text list support enables localization of the enumeration component identifiers and a representation of the symbolic component value in a text output in the visualization.

See also:

- PLC documentation: [Object DUT \[► 75\]](#)
- [Command Remove text list support \[► 1049\]](#)

17.16.11 Command Remove text list support

Symbol: 


Function: The command removes text list support from the selected enumeration object.

Call: Context menu of an enumeration object with text list support ().

Text list support enables localization of the enumeration component identifiers and a representation of the symbolic component value in a text output in the visualization.

17.17 Recipes

17.17.1 Command Add a new recipe

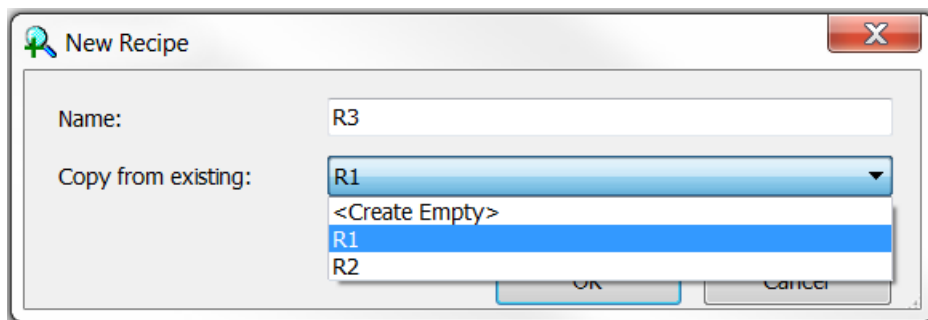
Symbol: 

Function: The command opens a dialog for inserting a new recipe (a new column) in the recipe definition.

Call: Menu **Recipes**, context menu


Requirement: A recipe definition is open in the editor.

When you run the command, a dialog opens where you can set the name of the new recipe. The dialog also offers an option for copying existing recipes into the new recipe.

**See also:**

- PLC documentation: [Changing values with recipes \[► 228\]](#)

17.17.2 Command Remove recipe

Symbol: 

Function: This command deletes a recipe from the currently open recipe definition.


Call: Menu **Recipes**, context menu

Requirement: A field is selected in the recipe column of a recipe definition.

See also:

- PLC documentation: [Changing values with recipes \[► 228\]](#)

17.17.3 Command Load Recipe

Symbol: 

Function: The command loads a recipe from a file.

Call: **Recipes** menu; context menu

Requirement: A field is selected in the recipe column of a recipe definition.

When you run the command, the standard dialog for selecting a file opens. The filter is automatically set to the file extension *.txtrecipe. After loading, the values of the selected recipe in the recipe definition are overwritten, and the display is updated.

If you want to overwrite only individual recipe variables with new values, remove the values for the other variables before loading the recipe into the recipe file. Entries without value specification are not read, which means that these variables are unaffected by the update on the controller and in the project. The following is an example of the entries in a recipe file. When it is loaded, only MAIN.nVar is written with a new value (6):


```
MAIN.nVar1:=
MAIN.nVar2:=6
MAIN.nVar3:=
MAIN.sVar4:=
MAIN.wsVar5:=
```

For values of the REAL/LREAL data type, the hexadecimal value is also written to the recipe file in some cases. This is necessary so that the exact identical value is restored during the back conversion process. In this case, change the decimal value and delete the hexadecimal value.

See also:

- PLC documentation: [Changing values with recipes \[► 228\]](#)

17.17.4 Command Save recipe

Symbol: 

Function: The command saves the values of recipe variables in a file.

Call: Menu **Recipes**, context menu

Requirement: A recipe value is selected in the recipe definition.

When the command is executed, TwinCAT saves the values of the selected recipe in a file with the extension *.txtrecipe, whose name must be defined. The standard dialog for saving a file opens. The format results from the settings of the Recipe Manager in the **Storage** tab.


● Overwriting the implicit recipe file

I The implicitly used recipe files, which are needed as a clipboard for reading and writing recipes, must not be overwritten. This means that the file name must be different to <Recipe name>.<Recipe definition name>.txtrecipe

See also:

- PLC documentation: [Changing values with recipes \[► 228\]](#)

17.17.5 Command Read Recipe

Symbol: 

Function: The command reads the variable values of a recipe from the controller.

Call: Menu **Recipes**, context menu


Requirement: The PLC project is in online mode, and a recipe value is selected in the recipe definition.

When the command is executed, TwinCAT overwrites the values of the selected recipe with the values read from the controller. In the process, the values are implicitly stored (in a file on the controller) and simultaneously displayed in the table of the recipe definition.

See also:

- PLC documentation: [Changing values with recipes \[► 228\]](#)

17.17.6 Command Write Recipe

Symbol: 

Function: The command writes the values of a recipe to the variables in the controller.

Call: Menu **Recipes**, context menu


Requirement: The PLC project is in online mode, and a recipe value is selected in the recipe definition.

When the command is executed, TwinCAT overwrites the values in the controller with the values of the selected recipe.

See also:

- PLC documentation: [Changing values with recipes \[► 228\]](#)

17.17.7 Command Load and Write Recipe

Symbol: 

Function: The command loads a recipe from a file and writes the values to the variables in the controller.

Call: **Recipes** menu; context menu

Requirement: The PLC project is in online mode, and a recipe value is selected in the recipe definition.

After running the command, you are asked whether the values from the file should also be written to the recipe in the project, or only to the PLC. An update of the values in the recipe may necessitate an Online Change with the next login.

When you execute the command, TwinCAT overwrites the values of the selected recipe in the recipe definition. In addition, the values of variables in the controller are overwritten with these recipe values.

If you want to overwrite only individual recipe variables with new values, remove the values for the other variables before loading the recipe into the recipe file. Entries without value specification are not read, which means that these variables are unaffected by the update on the controller and in the project. The following is an example of the entries in a recipe file. When it is loaded, only MAIN.nVar1 is written with a new value (6).


```
MAIN.nVar1:=  
MAIN.nVar2:=6  
MAIN.nVar3:=  
MAIN.sVar4:=  
MAIN.wstVar5:=
```

For values of the REAL/LREAL data type, the hexadecimal value is also written to the recipe file in some cases. This is necessary so that the exact identical value is restored during the back conversion process. In this case, change the decimal value and delete the hexadecimal value.

See also:

- PLC documentation: [Changing values with recipes \[► 228\]](#)

17.17.8 Command Read and Save Recipe

Symbol: 

Function: The command reads the variables values of a recipe from the controller and saves them in a file.

Call: Menu **Recipes**, context menu

Requirement: The PLC project is in online mode, and a recipe value is selected in the recipe definition.


After executing the command you are asked whether to read the variables values into the recipe in the project, or only save them. An update of the values in the recipe may necessitate an Online Change with the next login.

The values are saved with the default name for the recipe files, in accordance with the settings of the Recipe Manager (**Storage** tab).

See also:

- PLC documentation: [Changing values with recipes \[► 228\]](#)


17.17.9 Command Insert variable

Symbol: 

Function: The command adds a variable to the currently open recipe definition before the selected position.

Call: Menu **Recipes**, context menu


Requirement: The recipe definition is open in the editor, and simple view is selected.

TwinCAT adds the default text "NewVariable" in the **Variable** column. You must replace this name with the corresponding valid variable name. To this end, open the Input Assistant via the  button, or enter the variable name directly in the table field.

See also:

- PLC documentation: [Changing values with recipes \[► 228\]](#)

17.17.10 Command Remove variables

Symbol 

Function: The command removes the selected variable(s) from a recipe definition.


Call: [Del] key, context menu

Requirement: You have selected a variable.

See also:

- PLC documentation: [Changing values with recipes \[► 228\]](#)

17.17.11 Command Update structured variables

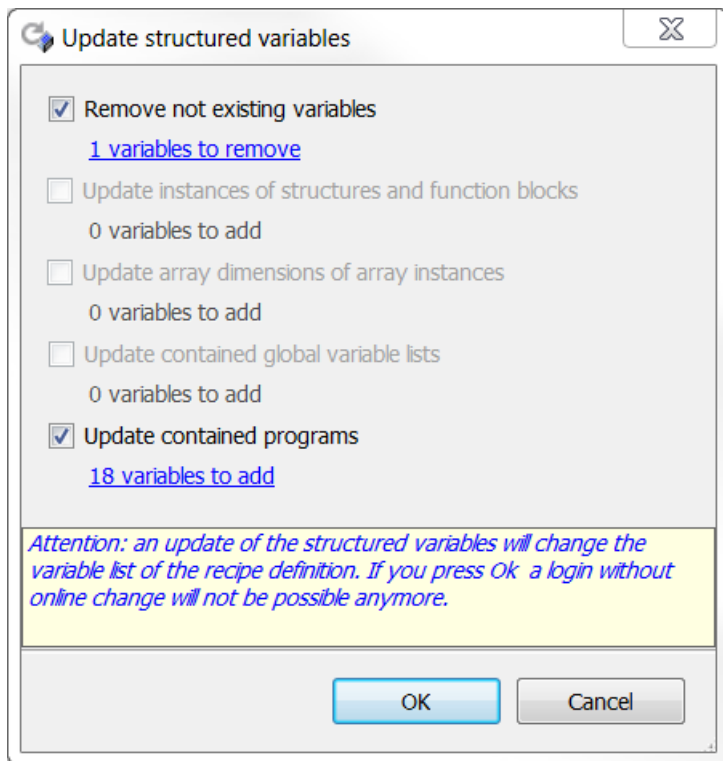
Symbol: 

Function: The command opens the dialog **Update structured variables**.

Call: Menu **Recipes**

In the dialog you can update the recipe definitions if the declaration of a structured variable or a function block has changed. If, for example, the dimension of an array has been changed, you can automatically remove or add the entries in the recipe definition.

Dialog Update structured variables



Remove non existing variables	: Variables that no longer exist in the project due a change in a structured element are removed from the recipe definition.
Update instances of structures and function blocks	: If the declaration of a structure or function block represented by an instance in the recipe definition is extended, the corresponding variables are added to the recipe definition.
Update array dimensions of array instances	: If the dimension of an array is extended, which is represented with an instance in the recipe definition, the corresponding variables are added to the recipe definition.
Update contained global variable lists	: If the declaration of a global variable list represented by an instance in the recipe definition is extended, the corresponding variables are added to the recipe definition.
Update contained programs	: If the declaration of a program is extended, which was instantiated in the recipe definition, the corresponding variables are added to the recipe definition.

See also:

- PLC documentation: [Changing values with recipes \[► 228\]](#)

17.17.12 Command Download recipes from the device

Symbol:

Function: The command initiates the synchronization of the recipes of the currently open recipe definition in the project with the recipes that are present on the device in the form of recipe files.

Call: Menu **Recipes**

Requirement: The PLC project is in online mode, and you have opened a recipe definition in the editor. In detail, the synchronization has the following effect:

- The current values for the recipe variables present in the project are overwritten with the values from the recipes on the controller. This may trigger an Online Change at the next login.
- If recipe variables are defined in the recipe files on the controller, which are missing in the recipe definition in the project, these variables are ignored on loading. For each recipe file, a message with the affected variables appears.
- If recipe variables are missing in the recipe files on the controller, which are included in the recipe definition in the project, for each recipe file a message appears showing the affected variables.
- If additional recipes were created on the controller for these variables, these are added to the recipe definition in the project.

17.18 Library

Command	Further information the PLC documentation
Library creation	
Command Save as library...	Command Save as library [► 270]
Command Save as library and install...	Command Save as library and install [► 271]
Library installation	
Command Library Repository	Library Repository [► 272]
Library management	
Dialog Library Manager	Library Manager [► 276]
Other commands and dialogs	
Command Add library	Command Add library [► 358]
Command Add library without resolution	Add library without placeholder resolution command [► 359]
Command Try reload library	Command Try reload library [► 360]
Command Delete library	Command Delete library [► 361]
Command Details	Command Details [► 361]
Command Dependencies	Command Dependencies [► 362]
Command Properties	Command Properties [► 362]
Command Placeholder	Placeholder [► 280]
Command Set to Effective Version	Command Set to Effective Version [► 364]
Command Set Always Newest Version	Command Set to Always Newest Version [► 364]

See also:


- PLC documentation: [Using libraries > Further commands and dialogs](#)

17.19 Visualization

The visualization commands are provided by the **Visual Editor** plug-in for the visualization commands menu category, which can be found in the dialog **Tools > Customize**. They enable you to edit a visualization object in the visualization editor.

Most commands are included in the **Visualization** menu as standard, and are therefore also available in the context menu of the visualization editor. If necessary, open the dialog **Tools > Customize**, in order to view or modify the menu configuration for the category **Visualization**.

17.19.1 Command Interface Editor

Symbol: 

Function: This command opens the Interface Editor for defining frame parameters in the visualization, which are intended to be referenced in the **Frame** element of another visualization. It is displayed in a tab view in the upper section of the Visualization Editor.

Call: Menu **Visualization**

17.19.2 Command Hotkey Configuration

Symbol: 

Function: This command opens the keyboard configuration editor for the current visualization. It is displayed in a tab view in the upper section of the Visualization Editor.

Call: Menu **Visualization**


17.19.3 Command Element List

Symbol: 

Function: This command opens the element list editor for the current visualization. It is displayed in a tab view in the upper section of the Visualization Editor.

Call: Menu **Visualization**


17.19.4 Command Align Left

Symbol: 

Function: This command aligns all selected visualization elements at the left edge of its leftmost element.

Call: Menu **Visualization**, context menu


17.19.5 Command Align Top

Symbol: 

Function: This command aligns all selected visualization elements at the top edge of its topmost element.

Call: Menu **Visualization**, context menu

17.19.6 Command Align Right

Symbol: 

Function: This command aligns all selected visualization elements at the right edge of its rightmost element.

Call: Menu **Visualization**, context menu

17.19.7 Command Align Bottom

Symbol: 

Function: This command aligns all selected visualization elements at the bottom edge of its bottommost element.

Call: Menu **Visualization**, context menu

17.19.8 Command Align Vertical Center

Symbol: 

Function: This command causes all selected visualization elements to be aligned with their common vertical center.

Call: Menu **Visualization**, context menu


17.19.9 Command Align Horizontal Center

Symbol: 

Function: This command causes all selected visualization elements to be aligned with their common horizontal center.

Call: Menu **Visualization**, context menu

17.19.10 Command Make horizontal spacing equal

Symbol: 

The command becomes active if three or more elements are selected.

1. Select all the elements to be positioned with the same horizontal spacing.
 - ⇒ The first element is highlighted in blue, the other elements are shown in gray.
2. Run the command **Make horizontal spacing equal**.
 - ⇒ The elements are positioned such that the leftmost and rightmost elements retain their positions and the elements in between are aligned with equal horizontal spacing.


17.19.11 Command Increase horizontal spacing

Symbol: 

The command becomes active if two or more elements are selected.

1. Select all the elements to be positioned with increased horizontal spacing.
 - ⇒ The first element is highlighted in blue, the other elements are shown in gray.
2. Run the command **Increase horizontal spacing**.
 - ⇒ The elements are positioned such that the blue element retains its position and the other elements are aligned horizontally with more spacing between the elements. The spacing increases by 1 pixel.

17.19.12 Command Decrease horizontal spacing


Symbol: 

The command becomes active if two or more elements are selected.

1. Select all the elements to be positioned with reduced horizontal spacing.
2. Run the command **Decrease horizontal spacing**.

- ⇒ The elements are positioned such that the blue element retains its position and the other elements are aligned horizontally with reduced spacing between the elements. The spacing decreases by 1 pixel.


17.19.13 Command Remove horizontal spacing

Symbol: 

The command becomes active if two or more elements are selected.

1. Select all the elements to be positioned without horizontal spacing.
 - ⇒ The first element is highlighted in blue, the other elements are shown in gray.
2. Run the command **Remove horizontal spacing**.
 - ⇒ The elements are positioned such that the blue element retains its position and the other elements are aligned horizontally without space between each other.


17.19.14 Command Make vertical spacing equal

Symbol: 

The command becomes active if two or more elements are selected.

1. Select all the elements to be positioned with the same vertical spacing.
 - ⇒ The first element is highlighted in blue, the other elements are shown in gray.
2. Run the command **Make vertical spacing equal**.
 - ⇒ The elements are positioned such that the topmost and bottommost elements retain their positions and the elements in between are aligned with equal vertical spacing.


17.19.15 Command Increase vertical spacing

Symbol: 

The command becomes active if two or more elements are selected.

1. Select all the elements to be positioned with increased vertical spacing.
 - ⇒ The first element is highlighted in blue, the other elements are shown in gray.
2. Run the command **Increase vertical spacing**.
 - ⇒ The elements are positioned such that the blue element retains its position and the other elements are aligned vertically with more spacing between the elements. The spacing increases by 1 pixel.


17.19.16 Command Decrease vertical spacing

Symbol: 

The command becomes active if two or more elements are selected.

1. Select all the elements to be positioned with reduced vertical spacing.
 - ⇒ The first element is highlighted in blue, the other elements are shown in gray.
2. Run the command **Decrease vertical spacing**.
 - ⇒ The elements are positioned such that the blue element retains its position and the other elements are aligned vertically with reduced spacing between the elements. The spacing decreases by 1 pixel.


17.19.17 Command Remove vertical spacing

Symbol: 

The command becomes active if two or more elements are selected.

1. Select all the elements to be positioned without vertical spacing.
 - ⇒ The first element is highlighted in blue, the other elements are shown in gray.
2. Run the command **Remove vertical spacing**.
 - ⇒ The elements are positioned such that the blue element retains its position and the other elements are aligned vertically without spacing between each other.


17.19.18 Command Make same width

Symbol: 

The command becomes active if more than one element is selected, except if a line or polygon element is selected.

1. Select all the elements that should have the same width.
 - ⇒ The first element is highlighted in blue, the other elements are shown in gray.
2. Run the command **Make same width**.
 - ⇒ All elements are assigned the width of the element marked in blue.


17.19.19 Command Make same height

Symbol: 

The command becomes active if more than one element is selected, except if a line or polygon element is selected.

1. Select all the elements that should have the same height.
 - ⇒ The first element is highlighted in blue, the other elements are shown in gray.
2. Run the command **Make same height**.
 - ⇒ All elements are assigned the height of the element marked in blue.


17.19.20 Command Make same size

Symbol: 

The command becomes active if more than one element is selected, except if a line or polygon element is selected.

1. Select all the elements that should have the same size.
 - ⇒ The first element is highlighted in blue, the other elements are shown in gray.
2. Run the command **Make same size**.
 - ⇒ All elements are assigned the size of the element marked in blue.

17.19.21 Command Size to Grid

Symbol: 

The command becomes active if more than one element is selected, except if a line or polygon element is selected.

1. Select all the elements whose position and size should be aligned to the grid.
 - ⇒ The first element is highlighted in blue, the other elements are shown in gray.
2. Run the command **Size to Grid**.
 - ⇒ All elements are aligned to the grid, according to their size and position.


17.19.22 Command Bring One to Front

Symbol: 

Function: This command positions the selected element one level higher, i.e. further in the foreground of the visualization. Elements at lower levels are concealed by those at higher levels.

Call: Menu **Visualization**, context menu

17.19.23 Command Bring to front

Symbol: 

Function: This command positions the selected element in the foreground of the visualization, i.e. at the highest level. Elements at lower levels are concealed by those at higher levels.

Call: Menu **Visualization**, context menu

17.19.24 Command Send One to Back

Symbol: 

Function: This command positions the selected element one level lower, i.e. further in the background of the visualization. Elements at lower levels are concealed by those at higher levels.

Call: Menu **Visualization**, context menu

17.19.25 Command Send to Back

Symbol: 

Function: This command positions the selected element in the background of the visualization, i.e. at the lowest level. Elements at lower levels are concealed by those at higher levels.

Call: Menu **Visualization**, context menu

17.19.26 Command Group

Symbol: 

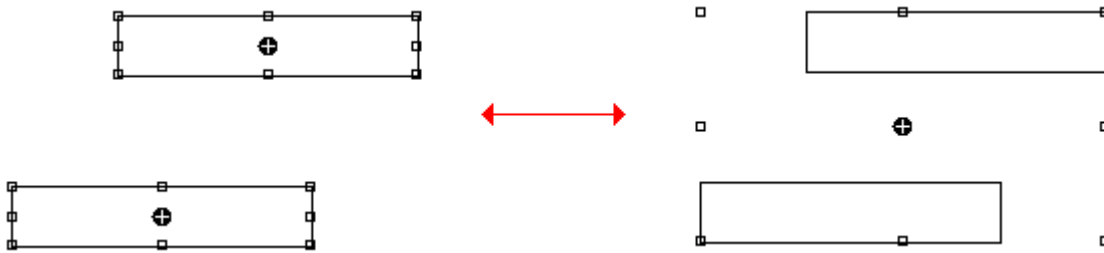
Function: This command groups the currently selected visualization elements and displays the group as a single object.

Call: Menu **Visualization**, context menu

Requirement: Several visualization elements are selected. For multiple selections hold down the [**<Shift>**] key while you click the items that you want. Alternatively, you can click in the editor window outside an element and draw a rectangle around the desired elements while pressing the mouse button.

Use the command **Ungroup** to break up the group.

The following diagram shows the grouping (from left to right) or ungrouping (from right to left) of two rectangle elements:



- [Command Ungroup \[▶ 1061\]](#)

17.19.27 Command Ungroup

Symbol:

Function: This command breaks up a group of visualization elements. The individual elements are shown individually selected again.

Call: Menu **Visualization**, context menu

See also:

- [Group \[▶ 1060\]](#)

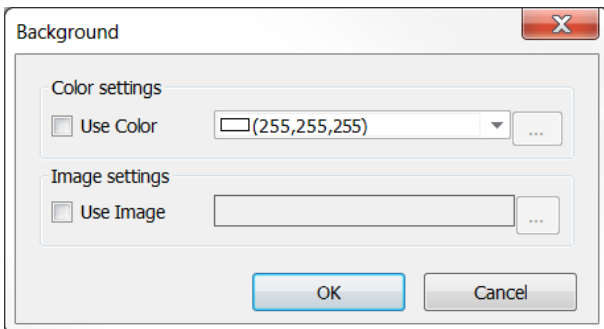
17.19.28 Command Background

Symbol:

Function: This command opens the dialog **Frame Configuration**, in which a color or an image for the visualization background can be selected.

Call: Menu **Visualization**, context menu

Dialog Background



Enable the desired option(s):

- **Bitmap:** To define a background image, enter the path of an image file, which is available in an image pool in the project. Enter the name of the image pool and the image file ID, separated by a dot ".": `<image pool>.<ID>` (e.g. "Images_1.drive_icon", "Images_1.43").
- **Color:** To define the background color of the visualization, select the desired color from the color list.

17.19.29 Command Select All

Symbol:

Function: This command selects all the elements of the visualization that is currently open in the editor.

Call: Menu **Visualization**, context menu

See also:

- [Deselect All \[▶ 1062\]](#)

17.19.30 Command Deselect All

Symbol: 


Function: This command clears the current selection of visualization elements.

Call: Menu **Visualization**

See also:

- [Command Select All \[▶ 1061\]](#)

17.19.31 Command Multiply visu element

Symbol: 

Function: The command opens the dialog **Multiply visu element**, which you can use to configure the multiplying procedure.

Call: Menu **Visualization**, context menu

Requirement: The visualization is active, and a template element is selected.

Dialog Multiply visu element

TwinCAT adds more elements, which resemble the template element. In this dialog you can configure the number and arrangement of the indices, as well as their replacement.

Tab Basic settings

Total number of elements:

TwinCAT inserts the new elements as a table. The number of lines is set in horizontal , the number of columns in vertical . The product of the two determines the actual total number of elements to be inserted.	
Horizontal	Number of elements per line Preset: Corresponds to the number of components in \$FIRSTDIM\$
Vertical	Number of elements per column Preset: Corresponds to the number of components in \$SECONDDIM\$

Offset between the elements:

TwinCAT arranges the elements in the visualization as a table. If you specify an offset, a spacing is inserted between the elements.

- 0 : The element frames overlap by one pixel
- 1 : The elements are in contact
- <n> : Between the elements there is a visible spacing of n-1 pixels

Horizontal	Line spacing of elements in pixels
Vertical	Column spacing of elements in pixels

Advanced Settings tab


First dimension:

Start index	Start index for \$FIRSTDIM\$ Preset: 1 means the specific index for \$FIRSTDIM\$ starts with 1. Example array[1, <\$SECONDDIM\$>]
Increment	Number by which the index is incremented Preset: 1

Second dimension:

Start index	Starting index for \$SECONDDIM\$ Preset: 1 means the specific index for \$SECONDDIM\$ starts with 1. Example array [<\$FIRSTDIM\$>, 1] are relevant
Increment	Number by which the index is incremented Preset: 1

17.19.32 Command Activate keyboard usage

Symbol: 

Function: This command is available in the menu bar for an integrated visualization (diagnostic visualization). It enables or disables keyboard operation in the online mode of a visualization.

Call: Menu **Visualization**

If keyboard operation is enabled, elements can be entered and selected via certain shortcuts. In this case, other keyboard commands are not executed as long as the visualization editor is active and online.

17.20 Miscellaneous

17.20.1 Command Implement interfaces

Function: The command updates the implemented interfaces for a function block by adding the interface elements that the function block does not currently contain.

Call: Context menu when the function block is selected in the PLC project tree.

Requirement: The function block implements an interface that you have changed. For example, you have added another method to the interface.

Applications

When this command is executed, the automatically created methods or properties are assigned a pragma attribute, which provokes compilation errors or warnings. This provides supports in the sense that automatically created elements do not remain empty unintentionally. Whether an error or warning attribute is used depends on the application.

Case 1:

Situation: The function block for which the command **Implement interfaces** is executed is **not** derived from another function block.

Consequence: When the command is executed, the interface elements are created in the function block without implementation ("stubs") and assigned a warning attribute (in the first line of the method/property declaration). The warnings generated during compilation alert you to the fact that these elements have been generated automatically and that you need to add the required implementation code.

```
{warning 'add method/property implementation'}
```

Procedure: Add the desired implementation code to the corresponding interface element (method or property). Then remove the warning attribute from the method or property declaration.

Case 2:

Situation: The function block for which the command **Implement interfaces** is executed is derived from another function block. The element (method or property) created when the command is executed in the derived function block is **not** provided by inheritance from the base function block (that is, the element does not exist under the base function block or a higher parent class).

Consequence/procedure: see case 1.

Case 3:

Situation: The function block for which the command **Implement interfaces** is executed is derived from another function block. The element (method or property) created when the command is executed in the derived function block is already provided by inheritance from the base function block (that is, the element exists under the base function block or a higher parent class).

Consequence: When the command is executed, the interface element is created in the derived function block without implementation ("stub") and assigned an error attribute (in the first line of the method/property declaration). The error generated during compilation alerts you to the fact that this interface element was generated automatically and that this method or property overwrites the corresponding element of the basic function block.

```
{error 'add method/property implementation or delete method/property to use base implementation'}
```

Procedure: If you want to overwrite or enhance the method or property of the basic function block, add the required implementation code to the element below the derived function block. Then remove the error attribute from the method or property declaration. However, if you do **not** want to overwrite the method or property of the basic function block, delete the method or property below the derived function block. In this case the method or property implementation of the basic function block is used.

17.21 Context menu TwinCAT project

17.21.1 Command Save <TwinCAT project name> as Archive...

Symbol: 

Function: The command opens the standard dialog for saving a file as an archive. The project can be stored under the desired path as a ***.tszip** archive.

Call: File menu, context menu

Requirement: The TwinCAT project is selected in the **Solution Explorer**.

Content of *.tszip	The *.tszip archive folder contains the TwinCAT project to be archived.
Command for opening	A tszip archive can be reopened with the following command: Command Project/Solution (Open Project/Solution) [► 862]
Note on PLC projects	If the TwinCAT project contains one or more PLC projects, the files and folders stored in the archive folder for those PLC projects will depend on the PLC project settings of the respective project. Settings tab [► 926]

17.21.2 Command Send by E-Mail...

Symbol: 

Function: The command starts the email program that is set in the system and opens a new email with the archive file of the selected project as an attachment.

Call: File menu, context menu

17.21.3 Command Backup <TwinCAT project names> automatically to the target system

Function: This command can be used to enable or disable automatic storage of the TwinCAT project as .tzip on the target system during activation. This is necessary if you want to load and open the project from the target system at a later time using the command [Command Open Project from Target](#) [▶ 869].

Call: Context menu of the TwinCAT project

Requirement: The TwinCAT project is selected in the Solution Explorer.

17.21.4 Command Compare <TwinCAT project name> with the target system...

Function: This command enables the selected TwinCAT project to be compared with the project status on the target system using the TwinCAT Project Compare tool.

Call: Context menu of the TwinCAT project

Requirement: The TwinCAT project is selected in the Solution Explorer and a target system is selected to which a TwinCAT project has already been downloaded.

See also:

- [Command Update project with target system...](#) [▶ 1065]

17.21.5 Command Update project with target system...

Function: This command enables the selected TwinCAT project to be updated to the project status of the connected target system. A pop-up dialog with a list of the changed files is opened for this purpose, which must be confirmed in order for the action to be successful. A detailed comparison is not provided here.

Call: Context menu of the TwinCAT project

Requirement: The TwinCAT project is selected in the Solution Explorer.

See also:

- [Command Compare <TwinCAT project name> with the target system...](#) [▶ 1065]

17.21.6 Command Load project with TwinCAT 2.xx Version...

Function: This command opens the standard browser dialog, through which a TwinCAT 2 System Manager file can be selected and imported. In this way you can convert an existing TwinCAT 2 project, including the System Manager settings and the PLC project, to TwinCAT 3.

Call: Context menu of the TwinCAT project

Requirement: The TwinCAT project has been newly created without changes and selected in the Solution Explorer.

See also:

- [Open a TwinCAT 2 PLC project](#) [▶ 57]


17.21.7 Command Show Hidden Configurations

Function: This command opens a detail menu in which the hidden configurations are listed and can be displayed again.

Call: Context menu of the TwinCAT project

Requirement: The TwinCAT project is selected in the Solution Explorer.

17.21.8 Command Remove From Solution


Symbol: 

Function: This command enables the TwinCAT project to be deleted from the Solution.

Call: Context menu of the TwinCAT project

Requirement: The TwinCAT project is selected in the Solution Explorer.

17.21.9 Command Rename


Symbol: 

Function: This command enables the TwinCAT project to be renamed in the **Solution Explorer**.

Call: Context menu of the TwinCAT project

Requirement: The TwinCAT project is selected in the Solution Explorer.

17.21.10 Command Build TwinCAT project

Symbol: 

Function: This command starts the compilation process or the code generation for the currently active TwinCAT project.

Call: **Build** menu if a TwinCAT project is currently selected, or context menu of the TwinCAT project

Requirement: The TwinCAT project is selected.

All of the projects (PLC, C++, etc.) contained in the TwinCAT project are compiled one after the other. The steps performed for a PLC project are described in section [Command Build PLC project \[► 929\]](#).

See also:

- [Command Rebuild a TwinCAT project \[► 1066\]](#)

17.21.11 Command Rebuild a TwinCAT project

Function: The command starts the compilation process or the code generation for the currently active TwinCAT project, even if it was last compiled without error.

Call: **Build** menu if a TwinCAT project is currently selected, or context menu of the TwinCAT project

Requirement: The TwinCAT project is selected.

When rebuilding the project, the TwinCAT project will first be cleaned (see also: [Command Clean TwinCAT project \[► 1066\]](#)) and subsequently built (see also: [Command Build TwinCAT project \[► 1066\]](#)).

17.21.12 Command Clean TwinCAT project

Function: This command deletes the local compilation information for the currently active PLC project and updates the language model of all objects.

Call: **Build** menu if a TwinCAT project is currently selected, or context menu of the TwinCAT project

Requirement: The TwinCAT project is selected.

All of the projects (PLC, C++, etc.) contained in the TwinCAT project are cleaned one after the other. The steps executed for a PLC project are described in the section [Command Clean PLC project \[► 930\]](#).

See also:

- [Command Rebuild a TwinCAT project \[► 1066\]](#)

17.21.13 Command Unload Project

Function: This command unloads the TwinCAT project so that all files of the TwinCAT project are released.

Call: Project menu or context menu of the TwinCAT project

Requirement: The TwinCAT project is selected in the Solution Explorer.

17.21.14 Import AutomationML via AML DataExchange...

Function: The command opens the standard browser dialog, via which you can search for and import a file in AutomationML format.

Call: The command can be called from the context menu of the TwinCAT project under **Import AutomationML** or via the **TwinCAT** item in the menu bar under **AutomationML** and **Import AutomationML**.

Requirement: The TwinCAT project is selected in the Solution Explorer.

See also

- Command: Open AML DataExchange Log (local)

17.21.15 Export AutomationML...

Function: The command opens the standard browser dialog for saving a file in AutomationML format for exporting the topology under the I/O node.

Call: The command can be called from the context menu of the TwinCAT project or via the **TwinCAT** item in the menu bar under **AutomationML**.

Requirement: The TwinCAT project is selected in the Solution Explorer.

18 PLC programming conventions

In the world of programming languages, there are several conventions that are observed depending on the tooling, focus and origin of the programming language, as well as the industry in which the application is used.

Based on the generally accepted programming guidelines (such as the MISRA C++ 2008 guidelines[...], the Java code conventions, the C# programming guide or PLCopen), the following section contains further programming information, which facilitates the creation of optimum, functional PLC program code.

Follow PLC programming conventions to achieve the following benefits:

- uniform structure of the PLC programs
- consistent naming of objects, variables and instances
- easy readability and intuitive comprehensibility of code and especially of the interfaces of function blocks, methods and functions
- simplified development, use and maintenance of the programs
- increased code quality through the systematic and early avoidance of error sources

New Beckhoff TwinCAT 3 PLC libraries, program samples and applications follow these "TwinCAT 3 PLC programming conventions".

Exceptions:

- Tc2_MC2_xxx libraries comply with the PLCopen programming convention.
- Tc2_xxx libraries comply with the original TwinCAT 2 libraries.
- Industry-specific libraries are based on the respective industry standard.
- Function blocks from the IEC61131 standard, which are part of the Tc2_Standard library.

Classification

The PLC programming conventions are divided into the following chapters:

- [Programming style \[► 1070\]](#)
- [Naming conventions \[► 1080\]](#)
- [Programming \[► 1089\]](#)

Within each chapter there are subchapters, which in turn contain a listing of topics.

The individual topics are classified as

- regulation, marked with [++], or as a
- recommendation, marked with [+].

You must implement topics classified as regulations. Recommendations should always be implemented, if possible, because they are advisable for several reasons.

Static code analysis

To reduce the risk of runtime errors, the code should at least be checked using the free variant of static code analysis ([Static Analysis Light \[► 150\]](#)). This is included from TwinCAT 3.1 Build 4018 and contains a minimal scope for code review. With the help of the full range of functions of TE1200 PLC Static Analysis, a more detailed review of the programming conventions is also possible (see the following section).

Verification of programming conventions with the help of TE1200 PLC Static Analysis

[TE1200 PLC Static Analysis](#) offers the possibility of versatile checks of PLC program code. With the help of these functionalities, compliance with many regulations, recommendations and naming conventions of the PLC programming conventions can be checked.

Subchapter Programming

Where available, each regulation/recommendation is accompanied by the corresponding rule from Static Analysis that can be used to verify compliance with the regulation/recommendation.

See also:

- Subchapter [Programming](#) [► 1089]
- Functionality [Rules](#) from TE1200 PLC Static Analysis

Subchapter Naming conventions

If available, the ID of the Static Analysis configuration field is given for each naming convention, in which the respective recommended name prefix must be entered. This allows most of the recommended prefixes for POU, DUTs, variables and instances to be checked by Static Analysis.

See also:

- Subchapter [Naming conventions](#) [► 1080]
- Functionality [Naming conventions](#) from TE1200 PLC Static Analysis

Pragmas and attributes

Also note the possibility to disable Static Analysis rules and naming conventions locally via pragma or attribute for checked locations where a specific rule/naming convention should no longer be reported (see also: [chapter Pragmas and Attributes](#) in TE1200 PLC Static Analysis). Ideally, you should comment on local deactivation with an appropriate explanation.

Download Static Analysis configuration file

1) Regulations and recommendations

The following link allows you to download a pre-built Static Analysis configuration file with the regulations [+ +] and recommendations [+] of the programming conventions enabled as rules. A violation of a regulation is output as an error after execution of the Static Analysis, a violation of a recommendation is classified as a warning.

https://infosys.beckhoff.com/content/1033/tc3_plc_intro/Resources/12663400331/.zip

2) Regulations, recommendations and naming conventions

The following link allows you to download a pre-built Static Analysis configuration file with the regulations [+ +], recommendations [+] and naming conventions of the programming conventions enabled as rules. A violation of a regulation or a naming convention is output as an error after execution of the Static Analysis, a violation of a recommendation is classified as a warning.

https://infosys.beckhoff.com/content/1033/tc3_plc_intro/Resources/12663402507/.zip

More rules

In addition to the rules that are explicitly mentioned within the following programming conventions, the TE1200 provides many other useful rules that can be used to detect careless errors or general programming errors at an early stage. These include, but are not limited to, the following rules.

- [SA0008: Check subrange types](#)
- [SA0041: Possibly loop-invariant code](#)
- [SA0044: Declarations with reference to interface](#)
- [SA0056: Constant out of valid range](#)
- [SA0059: Comparison operations always returning TRUE or FALSE](#)
- [SA0060: Zero used as invalid operand](#)
- [SA0062: Using TRUE and FALSE in expressions](#)
- [SA0140: Statements commented out](#)

18.1 Programming style

This section of PLC programming conventions covers the following topics.

Font and editor settings

1. [Use same font settings](#) [▶ 1070] [+]

Language

1. [Use English language](#) [▶ 1071] [+]
2. [Observe valid characters](#) [▶ 1071] [+]

Project structure

1. [Modular project structure](#) [▶ 1071] [+]
2. [Folder structure](#) [▶ 1071] [+]
3. [Sorting schema in the project tree](#) [▶ 1072] [+]
4. [No unused declarations/objects or useless code](#) [▶ 1072] [+]

Program structure

1. [Object-orientated programming](#) [▶ 1073] [+]
2. [Structure of program elements](#) [▶ 1073] [+]
3. [Structure of text blocks](#) [▶ 1073] [+]
4. [Comments](#) [▶ 1073] [+]

18.1.1 Font and editor settings

Topics:

1. [Use same font settings](#) [▶ 1070] [+]

Use same font settings

You should use identical font settings to ensure that the typeface is uniform across different programs. When TwinCAT 3 is integrated into Visual Studio, the font and tab width are preset by default.

Settings:

Tools > Options > Environment > Fonts and colors	
Font	Consolas
Font size	10
Tools > Options > TwinCAT > PLC Environment > Text editor	
Outline	Indent
Word wrap	No breaks
Tab width	4
Keep tabs	Yes
Indent width	4
Auto Indent	Smart with code completion

18.1.2 Language

Topics:

1. [Use English language \[► 1071\] \[+\]](#)
2. [Observe valid characters \[► 1071\] \[+\]](#)

Use English language

A PLC program (program code, variable names, comments etc.) is written entirely in English. American English is recommended here.

Observe valid characters

The names of program elements and variables may only contain the following characters, numbers and special characters:

- 0...9
- A...Z
- a...z
- Underscore character '_' as the only valid separator (more information below)

Double underscores ('__') should generally be avoided, because they are used for internal variables.

Use of the underscore character '_':

The underscore character '_' as the only valid separator is provided for the following designations:

- For objects between prefix and object name. Samples:
`FB_GetData, ST_BufferEntry, I_CylinderControl, E_StandardColor`
- Optional to implement a [sorting scheme \[► 1072\]](#) of objects in the project tree

18.1.3 Project structure

Topics:

1. [Modular project structure \[► 1071\] \[+\]](#)
2. [Folder structure \[► 1071\] \[+\]](#)
3. [Sorting schema in the project tree \[► 1072\] \[+\]](#)
4. [No unused declarations/objects or useless code \[► 1072\] \[+\]](#)

Modular project structure

You build up a TwinCAT 3 PLC project modularly. The folder structure should be guided by the various features and objects of a PLC project. If necessary, you sort the program elements according to a fixed scheme.

Folder structure

The folder structure of a TwinCAT 3 PLC project should be modular and based on the different functions/objects of a PLC project. You divide the PLC project into module folders on the first level. On the second level you can implement a finer modularization or an order according to element type (DUTs, ITFs, POUs etc.), depending on the complexity of the modules.

- Folder names of the first level: description of the functionality/module
- Folder names of the second level: further modularization or description of program elements

Sample:

Tc3_Plc_Project

- + [Topic X]
 - + DUTs
 - + ITFs
 - + POUs
- + [Topic Y]
 - + [Topic Y, Part a]
 - + DUTs
 - + POUs
 - + [Topic Y, Part b]
 - + ITFs
 - + POUs

Sorting schema in the project tree

The objects in the project tree are sorted alphabetically by TwinCAT 3. An optional scheme for sorting the objects in the project tree is therefore derived from the names of the objects.

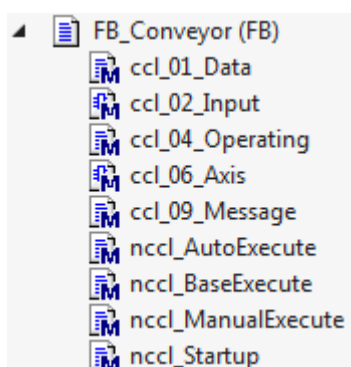
The sorting scheme can consist of a combination of numbers, key words or their abbreviations. It is located between the prefix and the actual name of the object and is separated from the prefix and the object name by an underscore. If the sorting scheme consists of several elements (e.g. numbers with keywords or their abbreviations), the respective subelements are separated from each other with underscore characters.

The sorting scheme should be annotated at a suitable place in the program element.

Alternatively, POU subfolders can be used to sort the subitems of a POU by topic.

Example of a sorting scheme:

The methods of the following function block are sorted using abbreviations and numbers. The abbreviations arrange the methods based on their call interval ('ccl' = cyclical; 'nccl' = noncyclical, e.g. event driven call); the numbers sort the cyclical methods according to their intended call sequence. These methods are provided as an example. The keywords for sorting are freely selectable; the underscores between keywords and numbers are optional (e.g. 'ccl01_Data' or 'ccl_01_Data').



No unused declarations/objects or useless code

Unused program elements in a project quickly lead to confusing program trees / code structures. During maintenance and extensions, the readability of the code can be greatly enhanced if the project only contains program elements that are actually used.

See also the topic [No unused declarations/objects or useless code \[► 1096\]](#) in the section [Programming \[► 1089\]](#).

18.1.4 Program structure

Topics:

1. [Object-orientated programming \[▶ 1073\] \[+\]](#)
2. [Structure of program elements \[▶ 1073\] \[+\]](#)
3. [Structure of text blocks \[▶ 1073\] \[+\]](#)
4. [Comments \[▶ 1073\] \[+\]](#)

Object-orientated programming

To take advantage of object-oriented programming, you should structure the PLC program in classes. Instead of a large number of function blocks, a selection of classes used with appropriate methods is used.

The benefits and user-friendliness of an object-oriented implementation should be assessed on an individual basis.

Structure of program elements

See: [Structure of program elements \[▶ 1074\]](#)

Structure of text blocks

See: [Structure of text blocks \[▶ 1076\]](#)

Comments

Avoid unnecessary comments as much as possible. Instead, the naming and program code should be self-explanatory, if possible, so that no further commenting is necessary. If a comment is required because it contributes significantly to understanding (e.g. units), note the following points:

Comments are intended to describe the use, benefits and content of program elements and their components and thus make programs quickly and easily understandable. They show up when viewing the program element and are also displayed in the form of a tooltip.

Comment place

If a comment is provided, it is located in the following places:

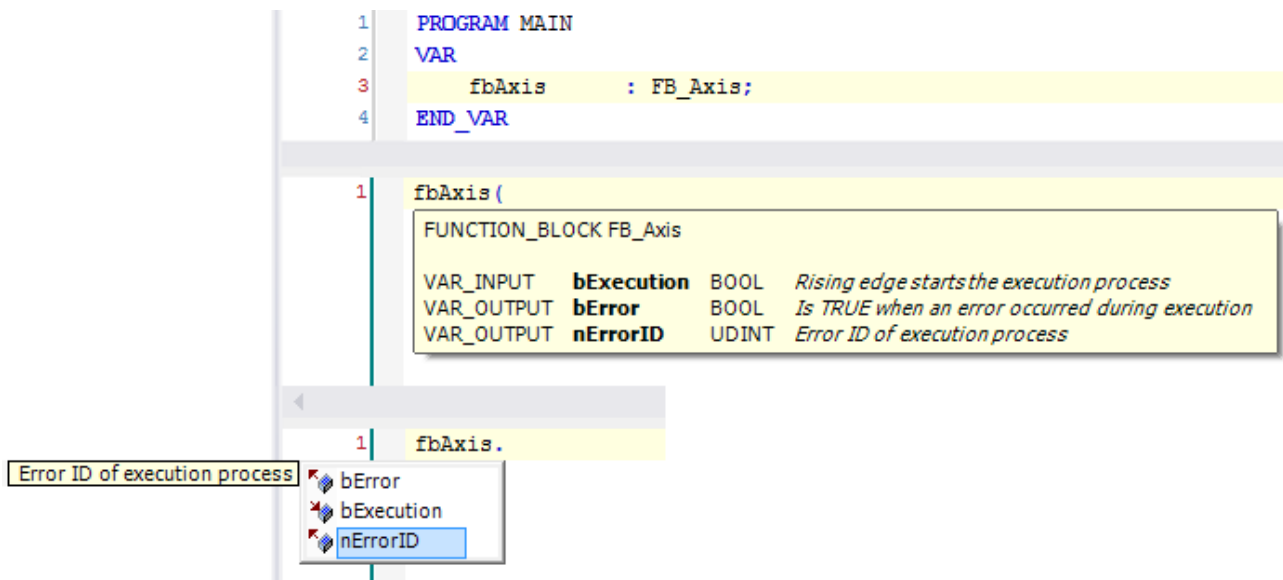
- Comments for program elements: above the first line of the element declaration (e.g. above the FUNCTION_BLOCK keyword)
- Comments for variables: in the same line following the declaration
- Declaration/program block: in the line above the block

Comment operator

- For single-line comments, the comment operator `//` is used.
- For comments that affect multiple lines, the comment operator `//` or the operator `(* ... *)` are used.

Sample

```
// This function block represents an axis
FUNCTION_BLOCK FB_Axis
VAR_INPUT
    bExecution      : BOOL;      // Rising edge starts the execution process
END_VAR
VAR_OUTPUT
    bError          : BOOL;      // Is TRUE when an error occurred during execution
    nErrorID        : UDINT;     // Error ID of execution process
END_VAR
```



18.1.4.1 Structure of program elements

Each new statement/declaration should start on a new line.

The following section explains the structuring of program elements (including POU, DUTs, GVLs) using text blocks, text sections and text regions.

Text blocks

- Thematically related declarations, assignments or calls with one or no parameter are grouped together in text blocks.
- Calls with more than one parameter, statements (such as IF or CASE) and loops are represented in separate text blocks.
- Each text block starts and ends with a blank line.
- Text blocks can be indented one level within a text region using the tab key, if you want to take advantage of being able to collapse the code below the region comment. If you use indentation, this should be implemented consistently within the POU.

Text sections

- Thematically related text blocks are bundled in text sections.
- Each text section starts and ends with a blank line.
- The first line of a text sections contains comments relating to the following text blocks.
- Text sections can be indented one level within a text region using the tab key, if you want to take advantage of being able to collapse the code below the region comment. If you use indentation, this should be implemented consistently within the POU.

Text regions

- Thematically related text sections are bundled in text regions.
- Each text region starts and ends with a dividing line within a comment.
- The first line below the separator contains comments relating to the following text sections, and therefore the text region. This region comment is on a level with the comment characters above.
- Region comments are not indented. If the text blocks and sections contained in the region are indented, all lines of the text region except for the region comment can thus be expanded or collapsed.

Sample

There are 2 text regions ("Drill settings" and "Conveyor settings") with 2 text sections ("Positions" and "Velocities"). In the text region "Drill settings", the text section "Velocities" consists of 2 text blocks (thematically related associations and an IF statement).

Program code:

```

//=====
// Drill settings

// Positions
fbDrill.nPositionLower := 100;
fbDrill.nPositionTop   := 500;

// Velocities
fbDrill.fVelocityRated := 40.0;
fbDrill.fVelocityMax   := 100.0;

IF fbDrill.fVelocityAverage > fbDrill.fVelocityRated THEN
    bWarning := TRUE;
    sWarning := 'Drill velocity: Average exceeded rated';
END_IF

//=====
// Conveyor settings

// Positions
fbConveyor.nPositionFilling := 50;
fbConveyor.nPositionDrill   := 200;
fbConveyor.nPositionDischarge := 300;

// Velocities
fbConveyor.fVelocityRated := 75.5;

//=====

```

Expanded text regions:

```

1 //=====
2 // Drill settings
3
4 // Positions
5 fbDrill.nPositionLower := 100;
6 fbDrill.nPositionTop   := 500;
7
8 // Velocities
9 fbDrill.fVelocityRated := 40.0;
10 fbDrill.fVelocityMax   := 100.0;
11
12 IF fbDrill.fVelocityAverage > fbDrill.fVelocityRated THEN
13     bWarning := TRUE;
14     sWarning := 'Drill velocity: Average exceeded rated';
15 END_IF
16
17 //=====
18 // Conveyor settings
19
20 // Positions
21 fbConveyor.nPositionFilling := 50;
22 fbConveyor.nPositionDrill   := 200;
23 fbConveyor.nPositionDischarge := 300;
24
25 // Velocities
26 fbConveyor.fVelocityRated := 75.5;
27
28 //=====

```

Collapsed text regions:

```

1 //=====
2 // Drill settings
3 [13 lines]
17 //=====
18 // Conveyor settings
19 [8 lines]
28 //=====

```

See also:

- Alternative possibility of realizing a folding capability: [Region pragma \[► 843\]](#)

18.1.4.2 Structure of text blocks

Text blocks within DUTs, GLVs and POU's consist of declarations, expressions, assignments, calls, statements or loops.

Declarations

- Declarations consist of 3 to 5 columns:
 - Variable name
 - [Allocation (e.g. AT %Q*)]
 - Data type
 - [Initialization]
 - [Comment]
- Whether the columns for allocation, initialization and comment exist depends on the respective program.
- Each declaration column within a program element starts at the same line level (example: all data types start at same line level).
- These variable declarations, if present, are created in a consistent order:

```

VAR_INPUT
END_VAR
VAR_IN_OUT CONSTANT
END_VAR
VAR_IN_OUT
END_VAR
VAR_OUTPUT
END_VAR

VAR
END_VAR
VAR_PERSISTENT
END_VAR
VAR_CONSTANT
END_VAR

```

- Each declared variable is given an identifier that is as self-explanatory as possible. If further information is needed to clarify the purpose or functionality of the variable, it is described by means of a comment following the declaration (on the same line).
- Thematically related declarations are
 - bundled in a declaration block,
 - described with a comment
 - and separated from other declaration blocks with a blank line.

Sample:

```

VAR
// Movement control
bExecuteMovements    AT%I*   : BOOL;          (* Controls the movement
                                         execution of car and bike *)
bMovementsDone       AT%Q*   : BOOL;          (* Indicates if the car and
                                         bike finished their movement *)

```



```

// General distance variables
nMovementTime      : INT;           // Elapsed movement time in seconds
nDistanceTotal     : INT;           // Total distance being covered by car and bike
nStartPosition     : INT           := 100; // General start position where car and bike be
gin to move

// Car
fbCar              : FB_Car;        // Car-FB whose movement is controlled
nDistanceCar      : INT;           // Distance being covered by car

// Bike
fbBike            : FB_Bike;        // Bike-FB whose movement is controlled
nDistanceBike     : INT;           // Distance being covered by bike
END_VAR

```

Expressions

- For expressions, each operator is separated by a space.
- Depending on the number and complexity of the subexpressions, parentheses can be used to visually illustrate the processing sequence and to enhance readability.

Examples:

```

nDistanceCar := fbCar.nStartPosition + (fbCar.nVelocity * nMovementTime);
nDistanceBike := fbBike.nStartPosition + (fbBike.nVelocity * nMovementTime);
nDistanceTotal := nDistanceCar + nDistanceBike;

```

Assignments

- Allocation operators for an assignment block start at the same line level.
- Equal line levels are achieved by the tab key.
- An allocation operator is followed by a space.
- Long assignments that exceed the screen width are split by means of a line break. The text part that ends up on a new line is indented (with the tab key) to a level after the allocation operator, e.g.:

```

bExpressionResult := bCondition1 AND (bCondition2 OR bCondition3)
                   AND bCondition4 AND (bCondition5 OR bCondition6);

```

- Thematically related assignments are
 - bundled in an assignment block
 - and separated from other blocks with a blank line.

Sample:

```

//=====
// Submodule enable

// Sensors
fbSensorEStop.bEnable := bGeneralEnable AND bSensorEnable;
fbSensorStartPos.bEnable := bGeneralEnable AND bSensorEnable;

// Cylinder
fbCylinderSystem1Main.bEnable := bGeneralEnable AND bCylinderEnable;
fbCylinderSystem1Sub.bEnable := bGeneralEnable AND bCylinderEnable;
fbCylinderSystem2Main.bEnable := bGeneralEnable AND bCylinderEnable;
fbCylinderSystem2Sub.bEnable := bGeneralEnable AND bCylinderEnable;

// Axes
fbAxisSubfoil.bEnable := bGeneralEnable AND bAxisEnable AND NOT bStop;
fbAxisMain.bEnable := bGeneralEnable AND bAxisEnable AND NOT bStop;

//=====

```

Calling methods/functions/function blocks

- Calls of methods/functions/function blocks with up to three parameters can be written single-line.
- Multiple single-line calls can be bundled into a text block.
- If the program elements have more than three parameters, they represent separate text blocks and are separated from other blocks by a blank line. Here the first parameter is in the line of the call or in the next line. The other parameters each start on a new line.
- The parameters beginnings are indented to the same level using the tab key.

- The allocation operators also start at the same line level.
- An allocation operator is followed by a space.
- If parameters are transferred to a function block or read at the same point in the program at which the instance is called, this occurs when the function block is called, and not immediately before or after the instance call.
- To increase the readability of the program code, a qualified call of methods/functions/function blocks is recommended. Here the assignment is formulated by explicitly naming the parameter name when it is called.

Negative sample:

```
fbTimerStart.IN := TRUE;
fbTimerStart.PT := T#10S;
fbTimerStart();
bStartExecution := fbTimerStart.Q;
```

Positive sample:

```
fbTimerStart( IN := TRUE,
              PT := T#10S,
              Q => bStartExecution);
```

Detailed sample:

```
//=====
// Stop

// Cylinder
fbCylinderPos1.Stop(bExecute := TRUE);
fbCylinderPos2.Stop(bExecute := TRUE);
fbCylinderPos3.Stop(bExecute := TRUE);

// Axes
fbAxis1.Stop(bExecute := TRUE, bDone => bAxis1Stopped);
fbAxis2.Stop(bExecute := TRUE, bDone => bAxis2Stopped);
//=====
```

Statements

- RETURN, CONTINUE, EXIT, JMP, IF, CASE
- JMP statement: jump statements can and should be avoided, since they reduce the readability of the code.
- IF statement:
 - An IF statement constitutes a separate text block and is separated from other blocks by a blank line.
 - Nested IF statements should be avoided, for better understanding and to improve the quality of software. Instead, a defined state should always apply, implemented using a CASE statement.
 - All IF-ELSIF statements should include an ELSE branch.
See also the topic [IF-ELSIF statements with ELSE branch \[► 1094\]](#) in the section [Programming \[► 1089\]](#).
 - The statements are indented one level, so that only the keywords IF/ELSIF/ELSE/END_IF are at the same level. The keyword THEN does not get its own line.
 - Long conditions that exceed the screen width are split by means of a line break. The text part that ends up on a new line is indented using the tab key to a suitable level, relative to the first line – either to the level of the IF keyword, or, for nested conditions, to the level of the corresponding / equivalent subcondition, e.g.:

Sample:

```
IF a > 10 THEN
  ...
ELSIF a < 10 THEN
  ...
ELSE
  ...
END_IF
```

```
IF bCondition1 AND bCondition2 AND (bCondition3 OR bCondition4)
  AND bCondition5 AND bCondition6
  AND (bCondition7 OR bCondition8 OR bCondition9 OR bCondition10) THEN
  ...
END_IF
```

• **CASE statement:**

- A CASE statement constitutes a separate text block and is separated from other blocks by a blank line.
- The enumeration values are separated from each other by a blank line and indented by one level relative to the CASE statement, so that only the keywords CASE/ELSE/END_CASE are at the same level.
- In addition, a comment about each value block can be useful.
- The statements for an enumeration value start 1-2 lines below the value (not directly next to the value) and are indented by one level relative to this value.

Sample:

```
//=====
// Cylinder sorting process

CASE eSortingState OF
  E_SortingState.DetectionOfBox:
    fbCylinder.MoveToBase();
    sVisuMessage := 'System in detection mode.';

    IF fbSensorDelay.bOut THEN
      eSortingState := eSorting_MoveCylToWorkPos;
    END_IF

  E_SortingState.MoveCylToWorkPos:
    fbCylinder.MoveToWork();

    IF fbCylinder.bAtWorkPos THEN
      eSortingState := eSorting_MoveCylToBasePos;
      sVisuMessage := '';
    ELSE
      sVisuMessage := 'Waiting for cylinder in work pos.';
    END_IF

  E_SortingState.MoveCylToBasePos:
    fbCylinder.MoveToBase();

    IF fbCylinder.bAtBasePos THEN
      eSortingState := eSorting_DetectionOfBox;
      sVisuMessage := '';
    ELSE
      sVisuMessage := 'Waiting for cylinder in base pos.';
    END_IF

ELSE
  sVisuMessage := 'System in non-existent state. Please check application.';
END_CASE

//=====
```

Loops (FOR, WHILE, REPEAT)

- The loop parameters are located in one line each (FOR ... DO, WHILE ... DO, REPEAT).
- The statements are indented by one level, so only the keywords FOR/END_FOR, WHILE/END_WHILE or REPEAT/UNTIL/END_REPEAT are at the same level.

Sample:

```
//=====
// Initialize storage areas with FOR loop

FOR nAreaCounter := cStorageStart TO cStorageEnd BY 1 DO
  aStorageAreas[nAreaCounter] := nAreaCounter;
END_FOR

//=====
// Initialize delivery areas with WHILE loop

nAreaCounter := cDeliveryStart;
```

```

WHILE nAreaCounter <= cDeliveryEnd DO
  aDeliveryAreas[nAreaCounter] := nAreaCounter;
  nAreaCounter                 := nAreaCounter + 1;
END_WHILE
//=====

```

18.2 Naming conventions

This section of PLC programming conventions covers the following topics.

General

1. [General name specifications \[▶ 1080\]](#) [+]
2. [Common abbreviations \[▶ 1081\]](#) [+]

Identifier

1. [Identifier for POU's and DUTs \[▶ 1083\]](#) [+]
2. [Identifiers for variables and instances \[▶ 1084\]](#) [+]

Global variable lists and parameter lists

1. [Global variable lists \[▶ 1087\]](#) [+]
2. [Parameter lists \[▶ 1088\]](#) [+]
3. [Use attribute 'qualified only' for GVL \[▶ 1088\]](#) [+]

Samples

See: [Samples \[▶ 1088\]](#)

18.2.1 General

Topics:

1. [General name specifications \[▶ 1080\]](#) [+]
2. [Common abbreviations \[▶ 1081\]](#) [+]

General name specifications

- Names are useful to facilitate understanding the purpose of the object.
- Avoid using the same names more than once. See also the topic [No multiple use of the same names \[▶ 1098\]](#) in the section [Programming \[▶ 1089\]](#).
- If an identifier is made up of several words, the first letter of each word is capitalized (CamelCase). Usually, no separators such as '_' are used between the words. If in exceptional cases good readability cannot be achieved using CamelCase, the use of the underscore character as a separator is recommended. For more exceptions to the use of the underscore character in object names, see the topic [Observe valid characters \[▶ 1071\]](#).
- Identifiers have a consistent prefix, so that the object type is easy to recognize. If a prefix is required for an identifier, note that this is case-sensitive. See also: [Identifier \[▶ 1083\]](#)
- Identifiers always begin with a letter.
- Abbreviations are also written in CamelCase. (Exceptions are stand-alone terms such as ID, CRLF, PC, PLC, These can be written in UpperCase)

Negative samples:

```

tADSTimeout      : TIME;
eCMDType         : E_CommandType;

```

Positive samples:

```
nAddr      : UDINT;  
nMsgCtrlMask : DWORD;  
cMaxCharacters : UDINT;  
tAdsTimeout : TIME;  
eCmdType    : E_CommandType;  
stRemotePCInfo : ST_RemotePCInfo;
```

Common abbreviations

The following common abbreviations may be used. No other abbreviation may be chosen for the corresponding term.

The list also contains terms which should not be abbreviated. In general, refrain from abbreviations that would shorten the term by less than 3 letters, as this does not improve readability. (Exceptions are the abbreviations Cnt and Idx.)

Abbreviation	Term
Abs	Absolute
Ack	Acknowledge
Act	Actual / Active (Current)
Addr	Address
/	Alarm
Auto	Automatic
Avg	Average
Bwd	Backward
/	Buffer
Calc	Calculation / Calculate
Char	Character
Cmd	Command
Com	Communication
Config	Configuration
Cnt	Count (Number of)
Dst	Destination
Diag	Diagnostic(s)
Diff	Difference
Dim	Dimension
/	Error
/	Execute
Fwd	Forward
Hdl	Handle
ID	Identifier
Idx	Index
In	Input
Info	Information
Init	Initialization / Initialize
Itf	Interface
Len	Length
Lib	Library
max	Maximum
Mem	Memory
min	Minimum
Msg	Message
Obj	Object
Op	Operation
Out	Output
Ptr	Pointer
Pos	Position
Prev	Previous
Rcv	Receive / Received
Ref	Reference
Src	Source
Srv	Server
sync	Synchronize / Synchronization
Temp	Temperature
Tmp	Temporary

Abbreviation	Term
/	Timeout
Velo	Velocity
Visualization	Visualization
/	Warning

18.2.2 Identifier

Topics:

1. [Identifier for POU and DUTs \[►_1083\] \[+\]](#)
2. [Identifiers for variables and instances \[►_1084\] \[+\]](#)

Identifiers for POU and DUTs

See: [Identifiers for POU and DUTs \[►_1083\]](#)

Identifiers for variables and instances

See: [Identifiers for variables and instances \[►_1084\]](#)

18.2.2.1 Identifiers for POU and DUTs

When naming POU and user-defined data unit types (DUTs), you should consider the following points.

- Prefixes of objects are always capitalized.
- The underscore character '_' is used as separator between the prefix and the actual object name.
- The actual object name always starts with a capital letter.
- If interfaces of TcCOM objects are directly embedded, they only have the prefix 'I', e.g. ITcUnknown.

Static Analysis:

Also note the option to [check programming conventions using TE1200 PLC Static Analysis \[►_1068\]](#).

Prefixes:

Object	Prefix	Description	Sample	Static Analysis Name Convention ID
FUNCTION_BLOCK	FB_	Function block	FB_WritePersistentData	NC0103
ACTION		Action (of a function block or a program)	MoveAbsolute	NC0106
METHOD		Method (of a function block or an interface)	Reset	NC0105
PROPERTY	according to the return type (see Identifiers for variables and instances [► 1084])	Property (of a function block, a program or an interface)	nErrorID fMotorTempC	NC0107 See also: Placeholder {datatype}
PROGRAM		Program	ModuleControl	NC0102
FUNCTION	F_	Function	F_MeterToInch	NC0104
STRUCT	ST_	Structure	ST_BufferEntry	NC0151
ENUM	E_	Enumeration type	E_MachineState E_Quality	NC0152
Type	T_	Alias type	T_Nibble	NC0154
UNION	U_	Union	U_Control	NC0153
INTERFACE	I_	Interface	I_Cylinder	NC0108
GVL	GVL as name or GVL_ as prefix	Global Variable List	GVL GVL_Axis GVL_Subsystem	
	GCL	Global constant list	GCL	
Param	Param as name or Param_ as prefix	Global parameter list	Param Param_Subsystem	

If a property <name> is directly represented by a variable of the function block, this variable is referred to as `_<name>`.

Enumeration:

When defining an enumeration, use the attribute `{attribute 'qualified_only'}`, which simplifies the use of the enumeration and at the same time makes it unnecessary to abbreviate the enumeration type. See also the topics [Use attributes 'qualified_only' and 'strict' for enumeration \[► 1101\]](#) in section [Programming \[► 1089\]](#).

```
{attribute 'qualified_only'}
{attribute 'strict'}
TYPE E_SignalStates :
(
    Red    := 0,
    Yellow,
    Green
);
END_TYPE
```

18.2.2.2 Identifiers for variables and instances

When naming variables and instances, consider the following points:

- Prefixes of variable and instance names are written in lower case.
- The first character after the prefix is capitalized (enable option **First character after prefix should be an upper case letter** of naming conventions from Static Analysis, see the following note on Static Analysis).

Static Analysis:

Also note the option to [check programming conventions using TE1200 PLC Static Analysis \[▶ 1068\]](#).

Prefixes:

1) Variables of elementary and dynamic data types:

Type	Prefix	Description	Declaration example	Call example	Static Analysis Name Convention ID
SINT, USINT, INT, UINT, DINT, UDINT, LINT, ULINT, BYTE, WORD, DWORD, LWORD, XINT, UXINT, XWORD	n	Integer, bit-based number	nErrorID : UDINT; nSize : UINT; nMask : WORD; nMessages : UINT;	nErrorID := 16#745; nSize := SIZEOF(ST_BufferEntry); nMask := nMask & 16#0FF0; nMessages := 3;	NC0005- NC0016, NC0035, NC0037, NC0038, NC0160
BOOL, BIT	b	boolean (bit)	bConfigured bReset		NC0003, NC0004
REAL, LREAL	f	float	fPosition		NC0017, NC0018
STRING	s	string	sNetID		NC0019
WSTRING	ws	wide string (unicode)	wsRouteName		NC0020
TIME, LTIME	t	time	tDelay		NC0021, NC0022
TIME_OF_DAY	td	time of day	tdClockTime		NC0025
DATE	d	date	dProductionStart		NC0023
DATE_AND_TIME	dt	date and time	dtProductionStart		NC0024
ARRAY[...] OF ...	a	array	aMessages		NC0030
POINTER TO ...	p	pointer	pData pNextProduct	pData := ADR(aBuffer);	NC0026
POINTER TO POINTER TO ...	pp	Nested pointer	ppData		
POINTER TO INTERFACE	pip	Pointer to an interface	pipCylinder		
REFERENCE TO ... 1) as method input 2) in other cases	1) according to the basic data type of the reference 2) ref	Reference	1) nData (with REFERENCE TO INT) 2) refSize	1) SampleMethod(nData := <...>) 2) refSize REF= nSize;	NC0027

• **Pointer:**

- Typed pointers are preferred over untyped pointers for safety reasons.
- Pointers should be declared as POINTER TO <data type>, where <data type> matches the data type of the variable to which the pointer refers.
- A pointer pointing to an array should be declared as POINTER TO ARRAY [...] OF <data type>, where <data type> matches the data type of the array to which the pointer refers.
- If the data type of the variable to which a pointer points is unknown and therefore cannot be typified, it should be declared as POINTER TO BYTE (or PVOID). The data type DWORD cannot be used as pointer for reasons of 64 bit compatibility.

- The content operator '^' is used for dereferencing a pointer.
- Particular attention should be paid to the validity of the address to which a pointer points.

Sample 1:

```
nSample      : INT;
pSampleToInt : POINTER TO INT;

pSampleToInt := ADR(nSample)
pSampleToInt^ := 123;           // Result: nSample = 123
```

Sample 2:

```
aSample      : ARRAY[1..3] OF LREAL;
pSampleToArrayOfLreal : POINTER TO ARRAY[1..3] OF LREAL;

pSampleToArrayOfLreal := ADR(aSample);
pSampleToArrayOfLreal^[2] := 4.56; // Result: aSample[2] = 4.56
```

• **Pointer arithmetic / pointer comparisons:**

- If possible, pointer arithmetic and the use of comparison operators on pointers should be avoided for safety reasons (e.g. increments such as [pSampleToLreal := pSampleToLreal + SIZEOF(LREAL)] or pointer comparisons using >, >=, <, <=).
- If you cannot do without these operations, use them at most within an array.
- It is particularly important to ensure that the pointers point to elements of the same array, and that the address range of the array is complied with.

2) Instances of objects, and user-defined data types:

Instances of function blocks that are defined in the IEC61131 standard and are of fundamental importance (R_TRIG, TON ...) have no other prefix. Example for correct naming:

```
fbAdsTimer : TON;
```

Type	Prefix	Description	Declaration example	Call example	Static Analysis Name Convention ID
FUNCTION_BLOCK	fb	Instance of a function block	fbWritePersData : FB_WritePersistentData;	fbWritePersData();	NC0031
STRUCT	st	Instance of a structure	stBufferEntry : ST_BufferEntry;	stBufferEntry.nCounter := 5;	NC0032
ENUM	e	Instance of an enumeration	eMachineState : E_MachineState; eQuality : E_Quality;	eMachineState := E_MachineState.Stop; eQuality := E_Quality.Good;	NC0029
TYPE (Alias)	according to the internal data type	Instance of an alias type	nNibble : T_Nibble; sNetId : T_AmsNetId;	nNibble := 16#1; sNetId := '1.2.3.4.5.6';	NC0033 See also: Placeholder {datatype}
INTERFACE	ip	Instance of an interface	ipCylinder : I_Cylinder;	ipCylinder := fbCylinderBase;	NC0036
UNION	u	Instance of a union	uCtrl : U_Control;	uCtrl.aRegister[1] := 16#02;	NC0034

3) Other:

Type	Prefix	Description	Declaration example	Static Analysis Note
Nested types	Only according to the outer type Exception 1: 'pp' for POINTER TO POINTER TO Exception 2: 'pip' for POINTER TO INTERFACE		pTelegramData : POINTER TO ARRAY[0..7] OF BYTE;	Disable option Recursive prefixes for combinable data types See also: Naming conventions (2)
Local and global constants	c		VAR CONSTANT cSyncID : UINT := 16#80; END_VAR VAR_GLOBAL CONSTANT cLowerThreshold : UDINT := 9; cOptionMove : DWORD := 16#04; END_VAR	NC0062, NC0070
Global variables		The same naming rules apply for global variables. (Do not add a second prefix to the variable names.)	VAR_GLOBAL nProducedUnits : UINT; END_VAR	Disable option Combine scope prefix with data type prefix See also: Naming conventions (2)
HRESULT	hr		hrErrorCode : HRESULT;	NC0160
PVOID	p	pointer	pData : PVOID;	NC0160
GUID		No prefix is assigned for GUID.		
AT%I*	optional before the data type prefix: I	allocated input	fActPos AT%I*: LREAL; oder: IfActPos AT%I*: LREAL;	
AT%Q*	optional before the data type prefix: O	allocated output	fSetPos AT%Q*: LREAL; oder: OfSetPos AT%Q*: LREAL;	

Allocated inputs or outputs can have an optional additional prefix 'I' or 'O' before the data type prefix. For instances, it represents the only prefix that is capitalized. The decision for or against the use of this optional prefix should be consistent, so that the name of allocated variables is uniform across the project.

18.2.3 Global variable lists and parameter lists

Topics:

1. [Global variable lists \[► 1087\] \[+\]](#)
2. [Parameter lists \[► 1088\] \[+\]](#)
3. [Use attribute 'qualified only' for GVL \[► 1088\] \[+\]](#)

Global Variable Lists

A list of global variables or constants is assigned the name "GVL" or the prefix "GVL_", followed by a description of the GVL in the name.

Examples:

- GVL
- GVL_Axis
- GVL_Subsystems

Parameter lists

A list of global parameters is given the name "Param" or the prefix "Param_", which is followed by a description of the parameter list in the name.

Examples:

- Param
- Param_Subsystems

Use attribute 'qualified_only' for GVL

When defining a [global variable list \[▶ 72\]](#) or a [parameter list \[▶ 73\]](#) use the attribute `{attribute 'qualified_only'}` [\[▶ 822\]](#), forcing the use of the GVL namespace when using the variables. By using the namespace (e.g.: `GVL_Ctrl.bPaintingActive`) the global scope of the variable becomes clear.

Positive sample:

Global variable list "GVL_Ctrl":

```
{attribute 'qualified_only'}
VAR_GLOBAL
    bPaintingActive : BOOL;
END_VAR
```

See also the topic [Use attribute 'qualified_only' for GVL \[▶ 1088\]](#) in section [Programming \[▶ 1089\]](#).

18.2.4 Samples

```
VAR_GLOBAL
    nSocketPort : INT;
    sNetID      : T_AmsNetId;
END_VAR

VAR_GLOBAL CONSTANT
    cSocketAddr : INT := 1;
    cLocalNetID : T_AmsNetId := '';
END_VAR

FUNCTION F_CompareVelocity : UINT
{attribute 'qualified_only'}
{attribute 'strict'}
TYPE E_StandardColor :
(
    Blue := 0,
    Green := 1
);
END_TYPE

FUNCTION_BLOCK FB_WirelessCom
VAR CONSTANT
    cFrequency : REAL := 868.0;
END_VAR
VAR
    eColor : E_StandardColor;
    aColors : ARRAY[1..cMaxColors] OF E_StandardColor;
END_VAR
-----
IF (sNetId = cLocalNetID) THEN
    eColor := E_StandardColor.Green;
    aColors[1] := eColor;
END_IF

FUNCTION_BLOCK FB_ModuleControl
VAR_IN_OUT
    aTerminalData : ARRAY[1..cOversamples] OF INT;
END_VAR
VAR
```

```
fbCalc      : FB_Calculate;  
END_VAR  
-----  
fbCalc.Reset();  
fbCalc( pDataIn      := ADR(aTerminalData),  
        nDataInSize := SIZEOF(aTerminalData) );
```

18.3 Programming

This section of PLC programming conventions covers the following topics.

Also note the option to check programming conventions using TE1200 PLC Static Analysis [[▶ 1068](#)].

General

Loops and conditions

1. CASE statement only with enumeration instance [[▶ 1092](#)] [++]
2. Declare limits of a loop as a constant [[▶ 1092](#)] [+]
3. Handle all enumeration values of the enumeration variables in CASE statement [[▶ 1093](#)] [+]
4. IF-ELSIF statements with ELSE branch [[▶ 1094](#)] [+]
5. Order of IF conditions [[▶ 1094](#)] [+]

Error Codes

1. Data type for error codes [[▶ 1095](#)] [++]
2. Error information to functions and methods [[▶ 1095](#)] [+]
3. Error information to function blocks [[▶ 1095](#)] [+]

Readability, maintainability

1. No unused declarations/objects or useless code [[▶ 1096](#)] [++]
2. No "magic numbers" [[▶ 1097](#)] [+]
3. No multiple use of the same names [[▶ 1098](#)] [+]
4. Same notation in declaration and implementation [[▶ 1098](#)] [+]
5. Avoid empty statements (;) [[▶ 1099](#)] [+]
6. Each assignment in separate line of code [[▶ 1099](#)] [+]

Libraries

Library development

1. Library names [[▶ 1100](#)] [++]
2. Revision control [[▶ 1100](#)] [++]
3. Encapsulation of internal type definitions [[▶ 1100](#)] [+]
4. Identifiers in libraries [[▶ 1100](#)] [+]

Use of libraries

1. Use of libraries [[▶ 1101](#)] [++]
2. Version check [[▶ 1101](#)] [++]
3. Fix library versions [[▶ 1101](#)] [+]

DUTs

Implementation of DUTs

1. Use attributes 'qualified only' and 'strict' for enumeration [[▶ 1101](#)] [++]

2. [Initialize no value, only the first value or all values for enumeration](#) [▶ 1102] [+]

Use of DUTs

1. [Assign enumeration variables to associated enumerators only](#) [▶ 1103] [++]

POUs

Implementation of functions, methods, actions

1. [No "call by value" of large parameters in functions/methods](#) [▶ 1104] [++]
2. [Do not declare large variables in functions/methods](#) [▶ 1105] [++]
3. [Do not use actions](#) [▶ 1105] [+]
4. [Use all parameters of a function/method internally](#) [▶ 1106] [+]
5. [Assign return value of a function/method only in one place](#) [▶ 1106] [+]
6. [Restrict access to methods as much as possible](#) [▶ 1107] [+]
7. [Grouping of parameters as structure](#) [▶ 1107] [+]

Use of functions, methods

1. [Evaluate returned error information of a POU](#) [▶ 1108] [++]
2. [Use return value of a function/method](#) [▶ 1108] [+]
3. [Do not call functions/methods within themselves](#) [▶ 1109] [+]

Implementation of function blocks

1. [Ensure online change capability](#) [▶ 1110] [++]
2. [Uniform interface with one-time asynchronous processing](#) [▶ 1110] [+]
3. [Uniform interface with continuous asynchronous processing](#) [▶ 1111] [+]
4. [Use all parameters of a function block internally](#) [▶ 1111] [+]
5. [Grouping of parameters as structure](#) [▶ 1111] [+]

Use of function blocks

1. [Do not declare function block instances as VAR PERSISTENT](#) [▶ 1112] [++]
2. [Evaluate returned error information of a POU](#) [▶ 1108] [++]
3. [Do not access local variables of a POU from outside](#) [▶ 1113] [++]
4. [No direct assignment of objects](#) [▶ 1113] [++]
5. [No temporary function block instances](#) [▶ 1114] [+]

Variables

General

1. [Do not test floating point numbers for equality or inequality](#) [▶ 1114] [++]
2. [Avoid unwanted results using explicit casting](#) [▶ 1115] [+]

Variable encapsulation

1. [Declare unchanged variables as VAR CONSTANT](#) [▶ 1116] [+]
2. [Do not shade global identifiers](#) [▶ 1117] [+]
3. [Restrict ADS access](#) [▶ 1117] [+]

Arrays

1. [Define array limits via constants](#) [▶ 1117] [++]
2. [Array lower limit of 1](#) [▶ 1118] [+]

Pointers, references, interfaces

1. [Temporary existence of pointers/references/interfaces to temporarily existing objects](#) [[▶ 1118](#)] [++]
2. [Reset pointer/references every cycle](#) [[▶ 1119](#)] [++]
3. [Check pointers/references/interfaces before each use](#) [[▶ 1119](#)] [++]
4. [Prefer references over pointers](#) [[▶ 1120](#)] [++]

Allocated variables

1. [Do not use direct addressing](#) [[▶ 1121](#)] [++]
2. [Avoid multiple write accesses to outputs](#) [[▶ 1121](#)] [++]
3. [Avoid overlapping memory areas of address variables](#) [[▶ 1122](#)] [+

Global variables

1. [Use attribute 'qualified only' for GVL](#) [[▶ 1123](#)] [+
2. [Use global variables wisely](#) [[▶ 1123](#)] [+

Strings

1. [Identifier for size and length specifications](#) [[▶ 1124](#)] [++]
2. [Recommended default size](#) [[▶ 1124](#)] [+
3. [Transfer of large strings](#) [[▶ 1124](#)] [+
4. [Processing of large strings](#) [[▶ 1125](#)] [+

Runtime behavior

General

1. [Intercept division by zero.](#) [[▶ 1125](#)] [++]

Dynamic memory

1. [Perform dynamic memory allocations carefully](#) [[▶ 1126](#)] [++]

Multiple tasks

1. [Avoid concurrent accesses to memory areas](#) [[▶ 1127](#)] [++]
2. [Avoid concurrent accesses to function blocks](#) [[▶ 1128](#)] [++]
3. [Use global variables wisely](#) [[▶ 1123](#)] [+

18.3.1 General

18.3.1.1 Loops and conditions

Topics:

1. [CASE statement only with enumeration instance](#) [[▶ 1092](#)] [++]
2. [Declare limits of a loop as a constant](#) [[▶ 1092](#)] [+
3. [Handle all enumeration values of the enumeration variables in CASE statement](#) [[▶ 1093](#)] [+
4. [IF-ELSIF statements with ELSE branch](#) [[▶ 1094](#)] [+
5. [Order of IF conditions](#) [[▶ 1094](#)] [+

CASE statement only with enumeration instance

In a CASE statement, an enumeration instance including corresponding enumerators should be used to define the state machine, instead of "magic numbers". This makes the values of the state machine self-speaking and no "magic numbers" are used.

Also note the following topics of the programming conventions:

- [Use attributes 'qualified_only' and 'strict' for enumeration](#) [► 1101]
- [No "magic numbers"](#) [► 1097]

Negative sample:

```
PROGRAM Sample_neg
VAR
  nState : INT; // Used to operate the sample state machine
END_VAR

// NON COMPLIANT: Integer variable and "magic values" are used to define state machine
CASE nState OF
  0: F_DoSomethingUsefulHere();
  1: F_DoSomethingUsefulHere();
  2: F_DoSomethingUsefulHere();
ELSE
  F_DoSomethingUsefulHere();
END_CASE
```

Positive sample:

```
// Enumeration for the positive sample
{attribute 'qualified_only'}
{attribute 'strict'}
TYPE E_SampleState :
(
  Entry, // Entrance part of a state
  Work, // Action part of a state
  Finish // Exit part of a state
);
END_TYPE

PROGRAM Sample_pos
VAR
  eState : E_SampleState; // Used to operate the sample state machine
END_VAR

// COMPLIANT: Enum instance and enum values are used to define state machine
CASE eState OF
  E_SampleState.Entry:
    F_DoSomethingUsefulHere();
  E_SampleState.Work:
    F_DoSomethingUsefulHere();
  E_SampleState.Finish:
    F_DoSomethingUsefulHere();
END_CASE
```

Declare limits of a loop as a constant

Constant variables should be used as limits of a loop. If an array is accessed within the loop, the array should be declared using the same constant variables.

- Lower limit of the array = lower limit of the loop = constant variable 1
- Upper limit of the array = upper limit of the loop = constant variable 2

Also note the following recommendations:

- [Define array limits via constants](#) [► 1117]
- [Array lower limit of 1](#) [► 1118]

Static Analysis:

Thematically recommended Static Analysis rules:

- [SA0072: Invalid uses of counter variable](#)
- [SA0080: Loop index variable for array index exceeds array range](#)

- [SA0081: Upper limit is not a constant](#)

General program elements for the following samples:

```

TYPE ST_Object :
STRUCT
  sName      : STRING;
  nID        : UINT;
END_STRUCT
END_TYPE

PROGRAM Sample
VAR CONSTANT
  cMin       : UINT := 1;
  cMax       : UINT := 10;
END_VAR
VAR
  aObjects   : ARRAY[cMin..cMax] OF ST_Object;
  bInitDone  : BOOL;
  nForIdx    : UINT;
END_VAR

```

Negative sample:

```

VAR
  nUpperBorder : UINT;
END_VAR

// Initialize objects
IF NOT bInitDone THEN
  nUpperBorder := cMin;

  FOR nForIdx := nUpperBorder TO 10 BY 1 DO
    aObjects[nForIdx].sName := CONCAT('Object_', UDINT_TO_STRING(nForIdx));
    aObjects[nForIdx].nID   := nForIdx;
  END_FOR

  bInitDone := TRUE;
END_IF

```

Positive sample:

```

// Initialize objects
IF NOT bInitDone THEN
  FOR nForIdx := cMin TO cMax BY 1 DO
    aObjects[nForIdx].sName := CONCAT('Object_', UDINT_TO_STRING(nForIdx));
    aObjects[nForIdx].nID   := nForIdx;
  END_FOR

  bInitDone := TRUE;
END_IF

```

Handle all enumeration values of the enumeration variables in CASE statement

In a CASE statement you should handle all enumeration values of the enumeration variables. If this does not make sense for a large number of identical or unused enumeration values, an ELSE branch can be used for these cases.

Static Analysis:

Verify using the following Static Analysis rules:

- [SA0075: Missing ELSE](#)
- [SA0076: Missing enumeration constant](#)

General program elements for the following samples:

```

// Enumeration for all samples in this rule
{attribute 'qualified_only'}
{attribute 'strict'}
TYPE E_ColorTrafficLight :
(
  Red      := 0,
  Yellow,
  Green
);
END_TYPE

```

```
PROGRAM Sample
VAR
    eColorTrafficLight : E_ColorTrafficLight; // Used to handle the state of the traffic light
END_VAR
```

Negative sample:

```
CASE eColorTrafficLight OF // NON COMPLIANT: enum value Green is not handled
    E_ColorTrafficLight.Red:
        SetLightRed();
    E_ColorTrafficLight.Yellow:
        SetLightYellow();
END_CASE
```

Positive sample:

```
CASE eColorTrafficLight OF // COMPLIANT: all enum values are handled
    E_ColorTrafficLight.Red:
        SetLightRed();
    E_ColorTrafficLight.Yellow:
        SetLightYellow();
    E_ColorTrafficLight.Green:
        SetLightGreen();
END_CASE
```

IF-ELSIF statements with ELSE branch

For safety reasons, it is advisable to add ELSE to an IF statement, if ELSIF is available. An ELSIF statement should be followed by ELSE, in order to demonstrate that all possible cases have been considered. If no statements are planned for the ELSE branch, a semicolon should follow after ELSE. During acceptance/commissioning of the program, a comment can be used to explain that the ELSE case was considered and why there are no statements for it.

General program elements for the following samples:

```
PROGRAM Sample
VAR
    nTest : INT:= 10; // Test value for sample
END_VAR
```

Negative sample:

```
IF nTest < 10 THEN // NON COMPLIANT: the ELSE should be used in any way
    nTest := 20;
ELSIF nTest > 20 THEN
    nTest := 10;
END_IF
```

Positive sample 1:

```
IF nTest < 10 THEN // COMPLIANT with any action in ELSE
    nTest := 20;
ELSIF nTest > 20 THEN
    nTest := 10;
ELSE
    F_DoSomethingUsefulHere(); // it's just a sample
END_IF
```

Positive sample 2:

```
IF nTest < 10 THEN // COMPLIANT with a comment in ELSE
    nTest := 20;
ELSIF nTest > 20 THEN
    nTest := 10;
ELSE
    ; (* No action needed for the case that nTest is >= 10 and <= 20 *)
END_IF
```

Order of IF conditions

Arrange the order of IF/ELSIF/ELSE conditions according to the probability of occurrence. Thus, the most likely case should be handled first. This improves both readability and, in some cases, performance because non-matching queries are processed less frequently.

Positive sample:

```

IF SUCCEEDED(hrErrorCode) THEN
  IF nReqIdx < cIdxMax THEN // requested index in range
    ; // handle request
  ELSIF nReqIdx = cIdxMax THEN // requested index in range but at upper limit
    ; // handle request
  ELSE
    ; // add error output (error caused by invalid index)
  END_IF
ELSE
  ; // add error handling (error in previous step)
END_IF

```

18.3.1.2 Error Codes

Topics:

1. [Data type for error codes \[► 1095\] \[++\]](#)
2. [Error information to functions and methods \[► 1095\] \[+\]](#)
3. [Error information to function blocks \[► 1095\] \[+\]](#)

Data type for error codes

Error codes should be specified and transferred as `hrErrorCode` (type `HRESULT`). The type `HRESULT` has the special feature that errors are represented by negative values. Warnings or information can optionally be output by means of positive values.

As an alternative you can specify `nErrorID` (type `UDINT`).

Declaration	Error range	No error	Message/info	Check functions
<code>hrErrorCode : HRESULT;</code>	<0	>=0	>0	<pre> IF SUCCEEDED(hrErrorCode) THEN ... END_IF IF FAILED(hrErrorCode) THEN ... END_IF </pre>
<code>nErrorID : UDINT;</code>	>0	0		<pre> IF nErrorID = 0 THEN ... END_IF IF nErrorID <> 0 THEN ... END_IF </pre>

Error information to functions and methods

Functions and methods do not require error information, provided that no error can occur. Sample: `c := AddTwoValues(a, b);`

Functions and methods can, in exceptional cases, output an error case using only Boolean, provided that there can be only one cause of the error.

In most cases functions and methods indicate an error by means of an error code (`HRESULT`) at the return value. Often a function block outputs the error of the last called method. See also the topic [Error information to function blocks \[► 1095\]](#).

Error information to function blocks

Function blocks do not require error information, provided that no error can occur.

It is recommended to provide error information to function blocks in the following way.

For small function blocks, which have few error causes, the following outputs are sufficient:

```

VAR_OUTPUT
  bError      : BOOL;          // TRUE if an error occurred.
  hrErrorCode  : HRESULT;      // '< 0' = error; '> 0' = info; '0' = no error/info
END_VAR

```

For more complex function blocks, which have many different error causes with possibly different error sources, a further output is recommended:

```

VAR_OUTPUT
  bError      : BOOL;          // TRUE if an error occurred.
  hrErrorCode  : HRESULT;      // '< 0' = error; '> 0' = info; '0' = no error/info
  ipErrorMessage : I_TcMessage :=
fbErrorMessage; // shows detailed information about occurred errors, warnings and more
END_VAR
VAR
  {attribute 'conditionalshow'}
  fbErrorMessage : FB_TcMessage;
END_VAR

```

`bError` enables the simplest possible query as to whether an error case exists. Also in the Online View of the PLC Editor you can easily see an error, if it is pending over several task cycles.

`hrErrorCode` specifies the error code and can be easily passed on as such. In the Online View of the PLC editor the hexadecimal value is displayed.

`ipErrorMessage` (alternatively `ipResultMessage`) shows detailed information about the occurred error. In the Online View of the PLC editor, among other things, a language-dependent plain text is displayed for error description. In addition, this message can be transmitted to the TwinCAT 3 EventLogger. Custom texts for custom error codes can be defined using custom event classes in event lists. For more information see the [documentation of the PLC library Tc3 EventLogger](#) and the [related sample](#). At the output of the function block, the information is provided as an interface to limit access to relevant content. However, it is recommended to ensure internally that the interface pointer is always valid.

On the topic of error codes, please also refer to the following topic on programming conventions:

- [Evaluate returned error information of a POU \[► 1108\]](#)

18.3.1.3 Readability, maintainability

Topics:

1. [No unused declarations/objects or useless code \[► 1096\] \[++\]](#)
2. [No "magic numbers" \[► 1097\] \[+\]](#)
3. [No multiple use of the same names \[► 1098\] \[+\]](#)
4. [Same notation in declaration and implementation \[► 1098\] \[+\]](#)
5. [Avoid empty statements \(;\) \[► 1099\] \[+\]](#)
6. [Each assignment in separate line of code \[► 1099\] \[+\]](#)

No unused declarations/objects or useless code

A project should contain no unexecutable paths, no "dead" (unnecessary) code, and no unused type declarations or variables.

Unused program elements in a project quickly lead to confusing code structures. During maintenance and extensions, the readability of the code can be greatly enhanced if the project only contains program elements that are actually used.

Static Analysis:

Verify using the following Static Analysis rules:

- [SA0001: Unreachable code](#)
- [SA0002: Empty objects](#)
- [SA0031: Unused signatures](#)

- [SA0032: Unused enumeration constants](#)
- [SA0033: Unused variables](#)
- [SA0035: Unused input variables](#)
- [SA0036: Unused output variables](#)

The rule [SA0033](#) is also included in the license-free variant [Static Analysis Light](#) [► 150].

Also note the possibility to disable Static Analysis rules and naming conventions locally via pragma or attribute for checked locations where a specific rule/naming convention should no longer be reported (see also: [chapter Pragmas and Attributes](#) in TE1200 PLC Static Analysis). Ideally, you should comment on local deactivation with an appropriate explanation.

For example, you can disable the rule [SA0002](#) for intended empty bodies of function blocks locally for this FB body by {analysis -2}.

No unexecutable paths

Negative sample:

```
IF FALSE THEN // NON COMPLIANT
    F_DoSomethingUsefulHere(); // will never be executed
END_IF
```

Positive sample:

```
IF bTest THEN // COMPLIANT
    F_DoSomethingUsefulHere(); // it's just a sample
END_IF
```

No "Dead" (unnecessary) code

Negative sample:

```
FUNCTION F_Sample_neg : INT
VAR_INPUT
    nA      : INT; // Variable a in term y = a*a + b
    nB      : INT; // Variable b in term y = a*a + b
END_VAR
VAR
    nTemp   : INT; // Used for temporary calculation of a*a
END_VAR
nTemp     := nA * nA; // NON COMPLIANT: nTemp will not be used later
F_Sample_neg := nA * nA + nB;
```

Positive sample:

```
FUNCTION F_Sample_pos : INT
VAR_INPUT
    nA      : INT; // Variable a in term y = a*a + b
    nB      : INT; // Variable b in term y = a*a + b
END_VAR
F_Sample_pos := nA * nA + nB; // COMPLIANT: no dead code in this sample
```

No "magic numbers"

Do not use "magic numbers".

Alternatively, constants or other fixed values can be used for comparisons or assignments. Texts such as output or comparison texts should be stored in constants, for example, and not defined directly in the source code.

Note also the following topic of the programming conventions:

- [CASE statement only with enumeration instance](#) [► 1092]

Negative sample:

```
PROGRAM Sample_neg
VAR
    nValue   : INT; // Sample INT-variable that is used for comparison
    bValueOk : BOOL; // Indicates if the value of sample INT-variable is OK
END_VAR
```

```
// NON COMPLIANT: A "magic value" is used for comparison
bValueOk := (nValue = 125);
```

Positive sample:

```
PROGRAM Sample_pos
VAR
    nValue      : INT;           // Sample INT-variable that is used for comparison
    bValueOk    : BOOL;        // Indicates if the value of sample INT-variable is OK
END_VAR
VAR_CONSTANT
    cValueOk    : INT := 125;   // Used to validate the sample INT-variable
END_VAR

// COMPLIANT: A constant variable is used for comparison
bValueOk := (nValue = cValueOk);
```

No multiple use of the same names

Avoid using the same names more than once. This includes the following cases:

- The name of an enumeration constant compared to the name of a variable
- Names of objects among each other
- The name of an object compared with the name of a variable

Static Analysis:

Verify using the following Static Analysis rules:

- [SA0013: Declarations with the same variable name](#)
- [SA0027: Multiple usage of name](#)

The rule [SA0027](#) is also included in the license-free variant [Static Analysis Light](#) [[▶ 150](#)].

Same notation in declaration and implementation

For reasons of uniformity, readability and maintainability, you should use the same notation of program elements and variables in the declaration and in the implementation.

Static Analysis:

Check with the help of Static Analysis rule:

- [SA0029: Notation in implementation different to declaration](#)

General program elements for the following samples:

```
FUNCTION F_Sample
```

Negative sample:

```
PROGRAM Sample
VAR
    bTest : BOOL;
END_VAR
-----
IF btest THEN
    f_Sample();
END_IF
```

Positive sample:

```
PROGRAM Sample
VAR
    bTest : BOOL;
END_VAR
-----
IF bTest THEN
    F_Sample();
END_IF
```

Avoid empty statements (;)

Avoid empty statements (;) so as not to expand the code unnecessarily. If an empty statement is absolutely necessary to clarify a particular case, the empty statement should be on a separate line. Additionally, you should comment why this program branch is empty.

Static Analysis:

Check with the help of Static Analysis rule:

- [SA0003: Empty statements](#)

General program elements for the following samples:

```
PROGRAM Sample
VAR
    nTest : INT := 10; // Test value used in sample
END_VAR
```

Negative sample:

```
IF nTest = 10 THEN
; // NON COMPLIANT without a useful comment
ELSE
    nTest := 10;
END_IF
```

Positive sample:

```
IF nTest = 10 THEN
; // COMPLIANT with a comment: In case that nTest is equal to 10, no action is
needed. This status is intended.
ELSE
    nTest := 10;
END_IF
```

Each assignment in separate line of code

Each assignment should be in a separate line of code. Thus, assignments should not be in conditions and allocation operators should not contain subexpressions.

Static Analysis:

Check with the help of Static Analysis rule:

- [SA0095: Assignments in conditions](#)

General program elements for the following samples:

```
PROGRAM Sample
VAR
    bVar1 : BOOL;
    bVar2 : BOOL;
    nA : INT;
    nB : INT;
    nC : INT;
END_VAR
```

Negative sample:

```
// NON COMPLIANT: assignment in condition
IF bVar1 := bVar2 THEN
    DoSomething();
END_IF

// NON COMPLIANT: more than one assignment in a single line
nA := nC * (nB := nC + nC);
```

Positive sample:

```
IF bVar1 = bVar2 THEN
    DoSomething();
END_IF

nB := nC + nC;
nA := nC * nB;
```

18.3.2 Libraries

18.3.2.1 Library development

Topics:

1. [Library names \[▶ 1100\] \[++\]](#)
2. [Revision control \[▶ 1100\] \[++\]](#)
3. [Encapsulation of internal type definitions \[▶ 1100\] \[+\]](#)
4. [Identifiers in libraries \[▶ 1100\] \[+\]](#)

Library names

Beckhoff PLC libraries start with the prefix **Tc** (e.g.: **Tc3_EventLogger**), have the file name extension ***.compiled-library** and are included in the library repository.

Choose library names that are short, yet unambiguous.

The namespace of a Beckhoff PLC library matches the name of the library.

The default placeholder of a Beckhoff PLC library matches the name of the library.

Revision control

Each PLC library has a four-digit version number (Major.Minor.Build.Revision).

Major	The first digit is incremented if the new version is incompatible with the previous version. Thereby the digits for minor and build are reset to one.
Minor	The second digit is incremented if the new version includes function expansion. In this case, the digit for Build is reset to one.
Build (Patch)	The third digit is incremented if the new version contains only bug fixes.
Revision	The fourth digit is always zero, if it is a release version.

If a PLC library is still in beta status, it is major version zero or a version number 0.x.y.z. A release version has at least the version number 1.1.1.0.

If other program parts provide a three-digit version (Major.Minor.Patch), the first three digits can be used accordingly (Build = Patch).

Encapsulation of internal type definitions

To prevent the use of internal type definitions from outside the library and to highlight to the user the POU and DUTs that are relevant to him, encapsulation is used in PLC libraries. The access specifier **INTERNAL** is recommended for this purpose.

The **PRIVATE** access specifier can also be used for individual methods of function blocks. To hide local variables of function blocks, the [Attribute 'conditionalshow_all_locals' \[▶ 799\]](#) can be used.

Identifiers in libraries

To prevent compiler errors to duplicate identifiers and to allow the library to be used without a mandatory namespace, the use of an additional abbreviation in the identifiers is recommended. This abbreviation of the library name is placed between the prefix and the actual name of the respective type.

Likewise, global variable lists of a library should be supplemented with the abbreviation to avoid multiple existence of lists named 'GVL'.

Samples (from the PLC library **Tc3_Filter**):

```
STRUCT ST_FTR_PT1
ENUM E_FTR_Type
FUNCTION_BLOCK FB_FTR_PT1
```



```
VAR_GLOBAL GVL_FTR
```

See also:

- [Use of libraries](#)

18.3.2.2 Using libraries

Topics:

1. [Use of libraries \[▶ 1101\]](#) [++]
2. [Version check \[▶ 1101\]](#) [++]
3. [Fix library versions \[▶ 1101\]](#) [+]

Use of libraries

PLC libraries are referenced as placeholders.

Details can be found in chapter [Library placeholders \[▶ 279\]](#) of the TwinCAT 3 PLC documentation.

Version check

PLC libraries contain a global constant of the type [ST_LibVersion \[▶ 791\]](#), where you can find information about the library version. This constant can be read during runtime and compared with a required library version, using the function `F_CmpLibVersion` (see `Tc2_System` library).

Fix library versions

When attaching a reference as a library placeholder, the library is attached with the version '* / newest version'.

As soon as an archive (e.g. PLC archive or TwinCAT archive) is created or the TwinCAT configuration is activated on the target system, however, the library versions should be fixed. The easiest way to do this is to use the 'Set to Effective Version' command in the context menu of the reference node.

Please also note the further information on the topic [Project delivery \[▶ 38\]](#).

See also:

- [Use of libraries](#)

18.3.3 DUTs

18.3.3.1 Implementation of DUTs

Topics:

1. [Use attributes 'qualified_only' and 'strict' for enumeration \[▶ 1101\]](#) [++]
2. [Initialize no value, only the first value or all values for enumeration \[▶ 1102\]](#) [+]

Use attributes 'qualified_only' and 'strict' for enumeration

When defining an enumeration, use the attributes [{attribute 'qualified_only'} \[▶ 822\]](#) and [{attribute 'strict'} \[▶ 823\]](#).

Static Analysis:

Verify using the following Static Analysis rules:

- [SA0025: Unqualified enumeration constants](#)
- [SA0171: Enumerations should have the 'strict' attribute](#)

Negative sample:

```
TYPE E_SignalColor :
(
  Red,
  Yellow,
  Green
);
END_TYPE
```

Positive sample:

```
{attribute 'qualified_only'}
{attribute 'strict'}
TYPE E_SignalColor :
(
  Red,
  Yellow,
  Green
);
END_TYPE
```

Initialize no value, only the first value or all values for enumeration

In an enumeration, you do not initialize any of the values, only the first one or all values with the allocation operator ':='.

Negative sample:

```
// NON COMPLIANT: init none, the first or all enumerators
{attribute 'qualified_only'}
{attribute 'strict'}
TYPE E_SignalColor :
(
  Red := 0,
  Yellow := 1,
  Green
);
END_TYPE
```

Positive sample 1:

```
// COMPLIANT: no enumerator is initialized
{attribute 'qualified_only'}
{attribute 'strict'}
TYPE E_SignalColor :
(
  Red,
  Yellow,
  Green
);
END_TYPE
```

Positive sample 2:

```
// COMPLIANT: first enumerator is initialized
{attribute 'qualified_only'}
{attribute 'strict'}
TYPE E_SignalColor :
(
  Red := 0,
  Yellow,
  Green
);
END_TYPE
```

Positive sample 3:

```
// COMPLIANT: all enumerators are initialized
{attribute 'qualified_only'}
{attribute 'strict'}
TYPE E_SignalColor :
(
```

```

    Red    := 0,
    Yellow := 1,
    Green  := 2
);
END_TYPE

```

See also:

- [DUT object](#)

18.3.3.2 Use of DUTs**Topics:**

1. [Assign enumeration variables to associated enumerators only](#) [[▶ 1103](#)] [++]

Assign enumeration variables to associated enumerators only

Only the enumerators of the associated enumeration should be assigned to the instance of an enumeration. To force this rule, it is recommended to set the attribute `{attribute 'strict'}` [[▶ 823](#)] when defining the type of the enumeration. See also: [Use attributes 'qualified_only' and 'strict' for enumeration](#) [[▶ 1101](#)]

Static Analysis:

Verify using the following Static Analysis rules:

- [SA0034: Enumeration variables with incorrect assignment](#)
- [SA0171: Enumerations should have the 'strict' attribute](#)

Negative sample:

```

{attribute 'qualified_only'}
TYPE E_ColorTrafficLight :
(
    Red    := 0,
    Yellow := 1,
    Green  := 2
);
END_TYPE

PROGRAM Sample_neg
VAR
    eColorTrafficLight : E_ColorTrafficLight; // Used to handle the state of the traffic light
END_VAR

CASE eColorTrafficLight OF
    E_ColorTrafficLight.Green:
        eColorTrafficLight := E_ColorTrafficLight.Yellow;
    E_ColorTrafficLight.Yellow:
        eColorTrafficLight := E_ColorTrafficLight.Red;
    E_ColorTrafficLight.Red:
        eColorTrafficLight := 2; // NON COMPLIANT: Only values defines in E_StandardCo
lor should be assigned to enum instance eColorTrafficLight
ELSE
    eColorTrafficLight := E_ColorTrafficLight.Red;
END_CASE

```

Positive sample:

```

{attribute 'strict'}
{attribute 'qualified_only'}
TYPE E_ColorTrafficLight :
(
    Red    := 0,
    Yellow := 1,
    Green  := 2
);
END_TYPE

```

```

PROGRAM Sample_pos
VAR
  eColorTrafficLight : E_ColorTrafficLight;           // Used to handle the state of the traffic
  light
END_VAR

CASE eColorTrafficLight OF
  E_ColorTrafficLight.Green:
    eColorTrafficLight := E_ColorTrafficLight.Yellow;
  E_ColorTrafficLight.Yellow:
    eColorTrafficLight := E_ColorTrafficLight.Red;
  E_ColorTrafficLight.Red:
    eColorTrafficLight := E_ColorTrafficLight.Green; // COMPLIANT
ELSE
  eColorTrafficLight := E_ColorTrafficLight.Red;
END_CASE

```

See also:

- [Object DUT \[► 75\]](#)

18.3.4 POU's

18.3.4.1 Implementation of functions, methods and actions

Topics:

1. [No "call by value" of large parameters in functions/methods \[► 1104\] \[++\]](#)
2. [Do not declare large variables in functions/methods \[► 1105\] \[++\]](#)
3. [Do not use actions \[► 1105\] \[++\]](#)
4. [Use all parameters of a function/method internally \[► 1106\] \[++\]](#)
5. [Assign return value of a function/method only in one place \[► 1106\] \[++\]](#)
6. [Restrict access to methods as much as possible \[► 1107\] \[++\]](#)
7. [Grouping of parameters as structure \[► 1107\] \[++\]](#)

No "call by value" of large parameters in functions/methods

Input and output parameters as well as the return value of a method are copied in most cases when the function/method is called, this is the "call by value". For parameters that occupy a lot of memory, this copy step takes more time. To write efficient program code, large parameters should be passed using "Call by reference". Thus, only one pointer is set. There are a few different possibilities for this, which are taken up in the sample. Please note that some options also allow write access to the input parameters.

See also "[Transferring large strings \[► 1124\]](#)".

Negative sample:

```

METHOD SampleMethod_neg : ST_DataCollection // return value with call by value
VAR_INPUT
  aSensor1Data : ARRAY[1..100] OF LREAL; // input with call by value
  aSensor2Data : ARRAY[1..200] OF LREAL; // input with call by value
  aSensor3Data : ARRAY[1..300] OF LREAL; // input with call by value
  aSensor4Data : ARRAY[1..400] OF LREAL; // input with call by value
END_VAR
VAR_OUTPUT
  aOutputData : ARRAY[1..500] OF LREAL; // output with call by value
END_VAR

```

Positive sample:

```

METHOD SampleMethod_pos : HRESULT
VAR_INPUT
  aSensor1Data : REFERENCE TO ARRAY[1..100] OF LREAL; // input with call by reference (write
access possible)
  pSensor2Data : POINTER TO ARRAY[1..200] OF LREAL; // input buffer, similar to call by
reference (write access possible)

```

```

    nSensor2DataSize : UDINT; // size in bytes of buffer to ensure the
correct buffer size
    aOutputData      : REFERENCE TO ARRAY[1..500] OF LREAL; // output with call by reference
    stDataCollection : REFERENCE TO ST_DataCollection; // output with call by reference
END_VAR
VAR_IN_OUT CONSTANT
    aSensor3Data : ARRAY[1..300] OF LREAL; // input with call by reference
END_VAR
VAR_IN_OUT
    aSensor4Data : ARRAY[1..400] OF LREAL; // input with call by reference (write
access possible)
END_VAR

```

Do not declare large variables in functions/methods

For variable declarations within a function/method, you should avoid a large number as well as a large data size. Such variable declarations are taken from stack memory. If functions/methods are called nested, the required amount of memory is added to the stack. There is not an unlimited amount of stack memory available and "stack overflow" situations should be avoided in advance.

The maximum size of the stack memory is specified in the TwinCAT project below the node system in the node Real-Time. Often this is 64 KB. Therefore, it is recommended to keep the total declared amount of data in a function/method as small as possible, for example below 1 KB. Therefore, especially with STRING types and arrays of arbitrary types, make sure that the lengths are as short as possible.

Alternatively, variables can be declared within the function block instance. Note here that these variables are permanently available and are not initialized each time the method is called.

The same applies to parameters of a function/method. See "[No "Call by value" of large parameters in functions/methods \[► 1104\]](#)".

Negative sample:

```

FUNCTION_BLOCK FB_Sample_neg
VAR
END_VAR

METHOD SampleMethod_neg : LREAL
VAR
    fSum      : LREAL;
    nCnt      : UINT;
    aLogList  : ARRAY[1..50] OF STRING(1023); // locally declared in method leads to stack allocation
END_VAR

```

Positive sample:

```

FUNCTION_BLOCK FB_Sample_pos
VAR
    aLogList : ARRAY[1..50] OF STRING(1023); // declared as member of the FB instance
END_VAR

METHOD SampleMethod_pos : LREAL
VAR
    fSum      : LREAL;
    nCnt      : UINT;
END_VAR

```

Positive sample (alternative):

```

FUNCTION_BLOCK FB_Sample2_pos
VAR
END_VAR

METHOD SampleMethod2_pos : LREAL
VAR
    fSum      : LREAL;
    nCnt      : UINT;
END_VAR
VAR_INST
    aLogList : ARRAY[1..50] OF STRING(1023); // declared as member of the FB instance
END_VAR

```

Do not use actions

A program or a function block should not contain actions. Implement methods instead. These can be defined as PUBLIC, PRIVATE or PROTECTED depending on their use.

Use all parameters of a function/method internally

All parameters of a function or a method should also be used internally.

Negative sample:

```
FUNCTION F_Sample_neg : INT
VAR_INPUT
  nA      : INT;          // Variable a in term y = a*a
  nB      : INT;          // NON COMPLIANT: nB will not be used
END_VAR

F_Sample_neg := nA * nA;
```

Positive sample:

```
FUNCTION F_Sample_pos : INT // COMPLIANT: no unused parameter
VAR_INPUT
  nA      : INT;          // Variable a in term y = a*a
END_VAR

F_Sample_pos := nA * nA;
```

Assign return value of a function/method only in one place

You should assign the return value of a function/method only in one place. An exception to this rule applies when the assignments are made in different branches of an IF or CASE statement.

Example 1:

Negative sample:

```
FUNCTION F_Sample_neg_1 : LREAL
VAR
  fOnePlusHalfEpsilon : LREAL;          // Auxiliary variable to calculate and validate the machine epsilon
END_VAR

// NON COMPLIANT: F_Sample_neg_1 is assigned or used more than once
F_Sample_neg_1 := 1.0;

REPEAT
  F_Sample_neg_1 := 0.5 * F_Sample_neg_1;          // NON COMPLIANT: see above
  fOnePlusHalfEpsilon := 1.0 + 0.5 * F_Sample_neg_1; // NON COMPLIANT: see above
UNTIL fOnePlusHalfEpsilon <= 1.0
END_REPEAT;
```

Positive sample:

```
FUNCTION F_Sample_pos_1 : LREAL
VAR
  fOnePlusHalfEpsilon : LREAL;          // Auxiliary variable to calculate and validate the machine epsilon
  fResult              : LREAL := 1.0;  // Temporary variable to
END_VAR

REPEAT
  fResult := 0.5 * fResult;
  fOnePlusHalfEpsilon := 1.0 + 0.5 * fResult;
UNTIL fOnePlusHalfEpsilon <= 1.0
END_REPEAT;

F_Sample_pos_1 := fResult;          // COMPLIANT: F_Sample_pos_1 is only assigned here
```

Example 2:

General program elements for this example:

```
// Enumeration for sample 2
{attribute 'qualified_only'}
{attribute 'strict'}
TYPE E_SampleStatus :
(
  Ok,          // Used when the status is OK
  Warning,    // Used when a warning occurs
  Error       // Used when an error occurs
);
END_TYPE
```

```
// GVL_Sample - global variable list for sample 2
VAR_GLOBAL CONSTANT
  cLimitMin    : INT := 10; // Minimal limit to validate a sample value
  cLimitMax    : INT := 20; // Maximal limit to validate a sample value
  cTolerance   : INT := 2;  // Upper and lower tolerance to validate a sample value
END_VAR

// Returns a status variable (ok, warning, error) for a measure value, using a constant range of values defined on a GVL
FUNCTION F_Sample : E_SampleStatus
VAR_INPUT
  nValue       : INT; // Sample value that is validated in respect of a range of values
END_VAR
```

Negative sample:

```
// NON COMPLIANT: F_Sample is assigned within different IF-
statements. Multiple write access on return value cannot be excluded.
IF (nValue >= cLimitMin) AND (nValue <= cLimitMax) THEN
  F_Sample := E_SampleStatus.Ok;
END_IF

IF (nValue >= (cLimitMin - cTolerance))
AND (nValue <= (cLimitMax + cTolerance)) THEN
  F_Sample := E_SampleStatus.Warning;
END_IF

IF (nValue < (cLimitMin - cTolerance))
OR (nValue > (cLimitMax + cTolerance))
  F_Sample := E_SampleStatus.Error;
END_IF
```

Positive sample:

```
// COMPLIANT: F_Sample is only assigned within one IF-
statement. No multiple write access on return value occurs.
IF (nValue >= cLimitMin) AND (nValue <= cLimitMax) THEN
  F_Sample := E_SampleStatus.Ok;
ELSIF (nValue >= (cLimitMin - cTolerance))
AND (nValue <= (cLimitMax + cTolerance)) THEN
  F_Sample := E_SampleStatus.Warning;
ELSE
  F_Sample := E_SampleStatus.Error;
END_IF
```

Restrict access to methods as much as possible

Access to methods should be restricted as much as possible using the PRIVATE or PROTECTED access modifiers. Thus, methods that are only intended to be called internally cannot be called externally. This helps to ensure that a function block is more likely to be used in the manner appropriate to its intended use. Encapsulation of methods therefore leads to safer code. In addition, subsequent changes to internal methods are easier if, as intended, they were not called by the user from the outside.

Grouping of parameters as a structure

If you have to specify many parameters when calling, it is a good idea to group them in one or more structures.

This has the advantage that default values are stored within the structure definition. Several constants of the structure data type can in turn provide different default values for different use cases.

The call can be implemented efficiently by passing the structure using "Call by reference". This avoids many individual "Call by value" transfers. See also the topic [No "Call by value" of large parameters in functions/methods \[► 1104\]](#).

See also:

- [Object Method \[► 90\]](#)
- [Object Function \[► 81\]](#)

18.3.4.2 Use of functions and methods

Topics:

1. [Evaluate returned error information of a POU \[► 1108\] \[++\]](#)
2. [Use return value of a function/method \[► 1108\] \[+\]](#)
3. [Do not call functions/methods within themselves \[► 1109\] \[+\]](#)

Evaluate returned error information of a POU

If a function, a method or a function block returns error information, always evaluate it.

Static Analysis:

Thematically recommended Static Analysis rules:

- [SA0009: Unused return values](#)
- [SA0169: Ignored outputs](#)

Negative sample:

```
PROGRAM Sample_neg
VAR
  fbFileOpen      : FB_FileOpen;      // FileOpen-FB for logger
  bFileOpenExec   : BOOL;             // Execute FileOpen-FB
END_VAR

// NON COMPLIANT: error information of fbFileOpen will not be used
fbFileOpen(
  sPathName := 'C:\TestFile.txt',
  nMode     := FOPEN_MODEWRITE OR FOPEN_MODETEXT,
  ePath     := PATH_GENERIC,
  bExecute  := bFileOpenExec,
  tTimeout  := T#3S);
```

Positive sample:

```
PROGRAM Sample_pos
VAR
  fbFileOpen      : FB_FileOpen;      // FileOpen-FB for logger
  bFileOpenExec   : BOOL;             // Execute FileOpen-FB
  bFileOpenError  : BOOL;             // Error flag of FileOpen-FB
  nFileOpenErrorID : UDINT;           // Error code of FileOpen-FB
END_VAR

// COMPLIANT: error information will be handled
fbFileOpen(
  sPathName := 'C:\TestFile.txt',
  nMode     := FOPEN_MODEWRITE OR FOPEN_MODETEXT,
  ePath     := PATH_GENERIC,
  bExecute  := bFileOpenExec,
  tTimeout  := T#3S,
  bError    => bFileOpenError,
  nErrId    => nFileOpenErrorID);

IF bFileOpenError THEN
  F_DoSomethingUsefulHere();          // Handle error here
END_IF
```

Use return value of a function/method

The return value of a function/method should be used, i.e. queried and evaluated, at the call point of the function/method. This is especially useful if an error value is returned in this way. However, exceptions are also possible where the return value does not have to be used in every call of the function/method.

Static Analysis:

Check with the help of Static Analysis rule:

- [SA0009: Unused return values](#)

General program elements for this rule:


```
(* Function for all samples in this rule: Adds a message to logger system.
  Returns TRUE if successful, FALSE on Error. *)
FUNCTION F_AddLogMessage : BOOL
VAR_INPUT
  sMessage      : WSTRING;      // Message to be logged
  dtTimestamp   : DATE_AND_TIME; // Timestamp of the message
END_VAR
```

Negative sample:

```
PROGRAM Sample_neg
VAR
  dtNow          : DATE_AND_TIME; // Actual system time
END_VAR

// NON COMPLIANT: return value of function will not be used
F_AddLogMessage(sMessage := "Test Message", dtTimestamp := dtNow);
```

Positive sample:

```
PROGRAM Sample_pos
VAR
  dtNow          : DATE_AND_TIME; // Actual system time
  bSendMessageOk : BOOL;         // Used to check if sending of message was successful
END_VAR

// COMPLIANT: return value of function will be used
bSendMessageOk := F_AddLogMessage(sMessage := "Test Message", dtTimestamp := dtNow);

IF NOT bSendMessageOk THEN
  F_DoSomethingUsefulHere(); // Handle error here
END_IF
```

Do not call functions/methods within themselves

Functions / methods should not call themselves directly or indirectly, in order to avoid recursions. In programming languages other than IEC61131, recursions should also be used advisedly.

Static Analysis:

Check with the help of Static Analysis rule:

- [SA0160: Recursive calls](#)

Negative sample:

```
FUNCTION F_Sample_neg : DWORD
VAR_INPUT
  nFac : DWORD; // The faculty of this value will be calculated
END_VAR

-----
IF nFac = 0 THEN
  F_Sample_neg := 1;
ELSE
  // NON COMPLIANT: implicit recursion. F_Sample_neg_IndirectFac calls F_Sample_neg
  F_Sample_neg := nFac * F_Sample_neg_IndirectFac(nFac := (nFac - 1));
END_IF

-----
FUNCTION F_Sample_neg_IndirectFac : DWORD
VAR_INPUT
  nFac : DWORD; // The faculty of this value will be calculated
END_VAR

-----
F_Sample_neg_IndirectFac := F_Sample_neg(nFac := nFac);
```

Positive sample:

```
FUNCTION F_Sample_pos : DWORD
VAR_INPUT
  nFac : DWORD; // The faculty of this value will be calculated
END_VAR
VAR
  nTemp : DWORD; // Temporary variable used to calculate faculty
  nCount : DWORD; // Counter variable used to calculate faculty
END_VAR

-----
nTemp := 1;

IF nFac > 0 THEN
```

```

FOR nCount := 1 TO nFac DO
    nTemp := nTemp * nCount;
END_FOR
END_IF
F_Sample_pos := nTemp;

```

See also:

- [Object Method \[► 90\]](#)
- [Object Function \[► 81\]](#)

18.3.4.3 Implementation of function blocks**Topics:**

1. [Ensure online change capability \[► 1110\] \[++\]](#)
2. [Uniform interface with one-time asynchronous processing \[► 1110\] \[++\]](#)
3. [Uniform interface with continuous asynchronous processing \[► 1111\] \[++\]](#)
4. [Use all parameters of a function block internally \[► 1111\] \[++\]](#)
5. [Grouping of parameters as structure \[► 1111\] \[++\]](#)

Ensure online change capability

The ability for online change is an important fundamental feature of PLC applications. A PLC program can be changed in various ways by means of online change without being stopped in the cyclic sequence. Often program code is changed in the implementation part, for which individual variables are also added in the declaration part.

Implement function blocks in such a way that instances are not affected in their functionality if they are moved in memory as a result of an online change.

For details refer to chapter [Execution of an online change \[► 251\]](#) in the TwinCAT 3 PLC documentation

Uniform interface with one-time asynchronous processing

In addition to the [naming conventions for variables \[► 1084\]](#) and the [order of text blocks for variable declarations \[► 1076\]](#), a uniform interface facilitates the use of function blocks.

For function blocks whose body call handles asynchronous and one-time ADS processing, you should use the following interface. If it is not an ADS communication, the parameter `sNetId` is omitted.

Processing starts on the rising edge of the input `bExecute`. This is only possible if the function block is not `bBusy`. Input parameters should not be changed by the user during an ongoing processing. This also ensures that the results of the processing (outputs) match the specified parameters (inputs).

When the processing is finished, `bBusy := FALSE` is set. At the latest now you have to set all output parameters. The user thus has the possibility of reacting to `bBusy = FALSE` and evaluating the outputs as a result.

```

FUNCTION_BLOCK FB_Sample
VAR_INPUT
    bExecute      : BOOL;
    ...           : ...;           // optional inputs
    ...           : ...;           // optional inputs
    tTimeout      : TIME := DEFAULT_ADS_TIMEOUT; // ADS communication timeout
    sNetId        : T_AmsNetId := ''; // keep empty '' for the local device
END_VAR
VAR_OUTPUT
    bBusy         : BOOL;
    bError        : BOOL;
    hrErrorCode    : HRESULT;

```

```

...      : ...;           // optional outputs
...      : ...;           // optional outputs
END_VAR

```

Uniform interface with continuous asynchronous processing

In addition to the [naming conventions for variables \[► 1084\]](#) and the [order of text blocks for variable declarations \[► 1076\]](#), a uniform interface facilitates the use of function blocks.

For function blocks whose body call handles asynchronous and continuous ADS processing, use the following interface. If it is not an ADS communication, the parameter `sNetId` is omitted.

Processing starts on the rising edge of `bEnable`. The function block operates as long as `bEnable` is set. Resetting `bEnable` does not necessarily result in the immediate termination of all processing due to the asynchronous processing. This is displayed by means of `bBusy = FALSE`. While `bEnable` is set, changed input parameters can be processed.

The output `bValid` indicates the successful processing and the validity of optional output parameters.

The output `bError` indicates an error that has occurred. More detailed information is provided by the error code at output `hrErrorCode`. Because `bError` can be set at any time during continuous processing and is often only present for one cycle, the user should query this output permanently.

```

FUNCTION_BLOCK FB_Sample
VAR_INPUT
  bEnable      : BOOL;
  ...          : ...;           // optional inputs
  ...          : ...;           // optional inputs
  tTimeout     : TIME := DEFAULT_ADS_TIMEOUT; // ADS communication timeout
  sNetId       : T_AmsNetId := ''; // keep empty '' for the local device
END_VAR
VAR_OUTPUT
  bValid       : BOOL;
  bBusy        : BOOL;
  bError       : BOOL;
  hrErrorCode  : HRESULT;
  ...          : ...;           // optional outputs
  ...          : ...;           // optional outputs
END_VAR

```

Use all parameters of a function block internally

All parameters of a function block should also be used internally.

Negative sample:

```

FUNCTION FB_Sample_neg
VAR_INPUT
  nA          : INT;           // Variable a in term y = a*a
  nB          : INT;           // NON COMPLIANT: nB will not be used
END_VAR
VAR_OUTPUT
  nY          : INT;
END_VAR
nY := nA * nA;

```

Positive sample:

```

FUNCTION FB_Sample_pos // COMPLIANT: no unused parameter
VAR_INPUT
  nA          : INT;           // Variable a in term y = a*a
END_VAR
VAR_OUTPUT
  nY          : INT;
END_VAR
nY := nA * nA;

```

Grouping of parameters as a structure

If you have to specify many parameters in the function block, it is a good idea to group them in one or more structures. For example, configuration parameters can be well separated visually from control parameters.

See also:

- [Object Function block \[► 84\]](#)

18.3.4.4 Use of function blocks**Topics:**

1. [Do not declare function block instances as VAR PERSISTENT \[► 1112\] \[++\]](#)
2. [Evaluate returned error information of a POU \[► 1112\] \[++\]](#)
3. [Do not access local variables of a POU from outside \[► 1113\] \[++\]](#)
4. [No direct assignment of objects \[► 1113\] \[++\]](#)
5. [No temporary function block instances \[► 1114\] \[++\]](#)

Do not declare function block instances as VAR PERSISTENT

Do not declare function block instances as [VAR PERSISTENT \[► 691\]](#) since this would cause the entire FB instance to be stored in the file of persistent variables. This could cause problems in function blocks that use ADS blocks or pointer variables internally, for example.

Evaluate returned error information of a POU

If a function, a method or a function block returns error information, always evaluate it.

Static Analysis:

Thematically recommended Static Analysis rules:

- [SA0009: Unused return values](#)
- [SA0169: Ignored outputs](#)

Negative sample:

```
PROGRAM Sample_neg
VAR
    fbFileOpen      : FB_FileOpen;      // FileOpen-FB for logger
    bFileOpenExec   : BOOL;             // Execute FileOpen-FB
END_VAR

// NON COMPLIANT: error information of fbFileOpen will not be used
fbFileOpen(
    sPathName := 'C:\TestFile.txt',
    nMode     := FOPEN_MODEWRITE OR FOPEN_MODETEXT,
    ePath     := PATH_GENERIC,
    bExecute  := bFileOpenExec,
    tTimeout  := T#3S);
```

Positive sample:

```
PROGRAM Sample_pos
VAR
    fbFileOpen      : FB_FileOpen;      // FileOpen-FB for logger
    bFileOpenExec   : BOOL;             // Execute FileOpen-FB
    bFileOpenError  : BOOL;             // Error flag of FileOpen-FB
    nFileOpenErrorID : UDINT;           // Error code of FileOpen-FB
END_VAR

// COMPLIANT: error information will be handled
fbFileOpen(
    sPathName := 'C:\TestFile.txt',
    nMode     := FOPEN_MODEWRITE OR FOPEN_MODETEXT,
    ePath     := PATH_GENERIC,
    bExecute  := bFileOpenExec,
    tTimeout  := T#3S,
    bError    => bFileOpenError,
    nErrId    => nFileOpenErrorID);
```

```
IF bFileOpenError THEN
    F_DoSomethingUsefulHere();           // Handle error here
END_IF
```

Do not access local variables of a POU from outside

Local variables of a POU should not be accessed from outside the POU.

In TwinCAT 3, write access to local variables from outside the programming object is not possible, as this would result in a compiler error. Read access, on the other hand, is not intercepted by the compiler.

In order to maintain the intended data encapsulation, it is strongly recommended not to access local variables of a POU from outside the POU – neither in read mode nor in write mode. Furthermore, with library function blocks it cannot be guaranteed that the local variables of a function block will remain unchanged during subsequent updates of the PLC library. This means that it is possible that the application project can no longer be compiled correctly after the library update.

Static Analysis:

Check with the help of Static Analysis rule:

- [SA0102: Access to program/fb variables from the outside](#)

Negative sample:

```
FUNCTION_BLOCK FB_Sample
VAR
    nLocal          : INT;
END_VAR

PROGRAM MAIN
VAR
    fbSample        : FB_Sample;
    nToBeProcessed  : INT;
END_VAR

-----
nToBeProcessed := fbSample.nLocal; // NON COMPLIANT: Do not access local variables from external context.
```

Positive sample:

```
FUNCTION_BLOCK FB_Sample
VAR_OUTPUT
    nOutput         : INT;
END_VAR
VAR
    nLocal          : INT;
END_VAR

PROGRAM MAIN
VAR
    fbSample        : FB_Sample;
    nToBeProcessed  : INT;
END_VAR

-----
nToBeProcessed := fbSample.nOutput; // COMPLIANT: Output variable is accessed from external context.
```

No direct assignment of objects

Simple data types (INT, BYTE, STRING, ...) or even structures are often assigned directly.

You should not assign instances of function blocks using allocation operator '='. When such an assignment is made, a copy of the object is made. However, function blocks often contain addresses internally, which would be corrupted if they were assigned.

To prohibit such an assignment in principle, you can set the [attribute 'no_assign'](#) [► 811] in the definition of a function block.

Static Analysis:

Check with the help of Static Analysis rule:

- [SA0014: Assignments of instances](#)

No temporary function block instances

You should not declare function block instances on the stack, i.e. not as temporary variables. Temporary instances are those that are declared in a method or a function or as VAR_TEMP, and are therefore reinitialized in each processing cycle or with each block call.

Function blocks have a state that is usually retained over several PLC cycles. An instance on the stack exists only for the duration of the function call. It is therefore only rarely useful to create an instance as a temporary variable. Secondly, function block instances are frequently large and require a great deal of space on the stack, which is usually limited on controllers. Thirdly, the initialization and often also the scheduling of the function block can take up quite a lot of time.

Static Analysis:

Check with the help of Static Analysis rule:

- [SA0167: Temporary function block instances](#)

The rule [SA0167](#) is also included in the license-free variant [Static Analysis Light \[► 150\]](#).

Negative sample:

```
FUNCTION F_Sample
VAR
    fbCtrl : FB_Control;
END_VAR
```

Positive sample:

```
FUNCTION F_Sample
VAR_INPUT
    fbCtrl : REFERENCE TO FB_Control;
END_VAR
```

See also:

- [Object Function block \[► 84\]](#)

18.3.5 Variables

18.3.5.1 General

Topics:

1. [Do not test floating point numbers for equality or inequality \[► 1114\] \[++\]](#)
2. [Avoid unwanted results using explicit casting \[► 1115\] \[+\]](#)

Do not test floating point numbers for equality or inequality

Floating point numbers should not be tested directly or indirectly for equality or inequality. Use the operators <, >, <=, >= instead.

Floating point numbers cannot always be represented without rounding error in a computer. Particularly in calculations with several floating point numbers, small rounding errors can quickly add up. Therefore, a floating point number should never be compared directly with the "=" operator. A measure for the smallest representable number is the so-called machine epsilon (ϵ). After the following examples a function is presented, which can be used to calculate the precision of a machine.

The only exception is the check for equal ('=') or unequal ('<>') to zero.

Static Analysis:

Check with the help of Static Analysis rule:

- [SA0054: Comparisons of REAL/LREAL for equality/inequality](#)

Negative sample 1:

```
PROGRAM Sample_neg_1
VAR
    fTest          : REAL    := 1E-20; // Test number
END_VAR

// The IF-condition is NON COMPLIANT: The comparison is not reliable due rounding errors
IF fTest = 0 THEN
    F_DoSomethingUsefulHere();        // it's just a sample
END_IF
```

Positive sample 1:

```
PROGRAM Sample_pos_1
VAR
    fTest          : REAL    := 1E-20; // Test number
END_VAR
VAR_CONSTANT
    cFloatEpsilon  : REAL    := 1E-12; // The deviation epsilon
END_VAR

// The IF-condition is COMPLIANT: Test with the deviation (± epsilon) around 0
IF (fTest > (0 - cFloatEpsilon)) AND (fTest < (0 + cFloatEpsilon)) THEN
    F_DoSomethingUsefulHere();        // it's just a sample
END_IF
```

Negative sample 2:

```
PROGRAM Sample_neg_2
VAR
    fCounter       : REAL;           // Test counter
END_VAR

// The WHILE-
condition is NON COMPLIANT: The comparison is not reliable due rounding errors. Attention: This loop
will be repeated endlessly!
WHILE fCounter <> 1 DO
    fCounter := fCounter + 0.001;
END_WHILE
```

Positive sample 2:

```
PROGRAM Sample_pos_2
VAR
    fCounter       : REAL;           // Test counter
END_VAR

// The WHILE-
condition is COMPLIANT because of comparison operator '<'. The loop ends when fCounter is not smaller
than 1 anymore.
WHILE fCounter < 1 DO
    fCounter := fCounter + 0.001;
END_WHILE
```

The following function can be used to calculate the precision of a machine.

```
FUNCTION F_CalculateMachineEpsilon : LREAL
VAR
    fOnePlusHalfEpsilon : LREAL;           // Auxiliary variable to calculate and validate the
machine epsilon
    fResult              : LREAL := 1.0;   // Temporary variable to calculate the result
END_VAR

REPEAT
    fResult := 0.5 * fResult;              // Devide fResult by 2.
    fOnePlusHalfEpsilon := 1.0 + 0.5 * fResult; // If new result divided by 2 plus 1.0 is smaller th
an or equal to 1.0, the Epsilon is calculated.
UNTIL fOnePlusHalfEpsilon <= 1.0
END_REPEAT;

F_CalculateMachineEpsilon := fResult;     // Set return value to fResult
```

Avoid unwanted results using explicit casting

Due to implicit or missing casting, undesired results may occur. On the one hand, this can be caused by too late implicit conversion of the compiler (see sample and [SA0130](#)). On the other hand, undesired results can occur by casting variables too late, whose operation is then executed in the platform-dependent register width of the processor and not in the size of the variable type (see [SA0066](#)). Therefore, when programming, attention should be paid to the (size of the) variable types and, if necessary, a conversion should be performed using explicit casting.

Static Analysis:

Verify using the following Static Analysis rules:

- [SA0020: Possibly assignment of truncated value to REAL variable](#)
- [SA0066: Use of temporary results](#)
- [SA0130: Implicit expanding conversions](#)

General program elements for the following samples:

```
PROGRAM Sample
VAR
  nLINT   : LINT;
  nDINT   : DINT;
END_VAR
```

Negative sample:

```
// Possibly wrong result due to variable overflow caused by late converting of compiler
nLINT := nDINT * nDINT;
```

Positive sample:

```
// Correct result due to explicit variable cast
nLINT := TO_LINT(nDINT) * TO_LINT(nDINT);
```

18.3.5.2 Variable encapsulation**Topics:**

1. [Declare unchanged variables as VAR CONSTANT \[► 1116\] \[+\]](#)
2. [Do not shade global identifiers \[► 1117\] \[+\]](#)
3. [Restrict ADS access \[► 1117\] \[+\]](#)

Declare unchanged variables as VAR CONSTANT

Variables that are not modified should be declared as VAR CONSTANT.

Static Analysis:

Check with the help of Static Analysis rule:

- [SA0012: Variables which could be declared as constants](#)

Negative sample:

```
PROGRAM Sample_neg
VAR
  fTest       : REAL := 1E-20; // Test value
  cFloatEpsilon : REAL := 1E-12; // NON COMPLIANT: cFloatEpsilon will not be changed, but is not
  declared as VAR CONSTANT
END_VAR
-----
IF (fTest > (0 - cFloatEpsilon)) AND (fTest < (0 + cFloatEpsilon)) THEN
  F_DoSomethingUsefulHere(); // it's just a sample
END_IF
```

Positive sample:

```
PROGRAM Sample_pos
VAR
  fTest       : REAL := 1E-20; // Test value
END_VAR
VAR CONSTANT
  cFloatEpsilon : REAL := 1E-12; // COMPLIANT: cFloatEpsilon will not be changed and is declared
  as VAR CONSTANT
END_VAR
-----
IF (fTest > (0 - cFloatEpsilon)) AND (fTest < (0 + cFloatEpsilon)) THEN
  F_DoSomethingUsefulHere(); // it's just a sample
END_IF
```


Do not shade global identifiers

Identifiers in an inner relationship should not "shade" more global identifiers. Identifiers in an inner relationship include variables that are instantiated in a method. In this case, more global identifiers are variables of the corresponding function block, for example.

Static Analysis:

Verify using the following Static Analysis rules:

- [SA0013: Declarations with the same variable name](#)
- [SA0027: Multiple usage of name](#)

The rule [SA0027](#) is also included in the license-free variant [Static Analysis Light](#) [► 150].

General program elements for the following samples:

```
FUNCTION_BLOCK FB_Sample
VAR
    sDescription : STRING; // STRING for POU description
END_VAR
-----
sDescription := 'This is a function block';
```

Negative sample:

```
METHOD PUBLIC nccl_Test : STRING
VAR
    sDescription : STRING; // NON COMPLIANT: sDescription is already defined in method's outer scope
e FB_Sample
END_VAR
-----
sDescription := 'This is a method';
nccl_Test := sDescription;
```

Positive sample:

```
METHOD PUBLIC nccl_Test : STRING
VAR
    sMethodDescription : STRING; // COMPLIANT
END_VAR
-----
sMethodDescription := 'This is a method';
nccl_Test := sMethodDescription;
```

Restrict ADS access

If required, access to variables can be restricted so that there is no visibility via ADS. Such variables are then not usable for HMIs. For this purpose, the [attribute 'TcNoSymbol'](#) can be used.

18.3.5.3 Arrays

Topics:

1. [Define array limits via constants](#) [► 1117] [++]
2. [Array lower limit of 1](#) [► 1118] [+]

Define array limits via constants

You should define the limits of an array using constant variables. When accessing the array within a loop, the same constant variables should be used to define the loop limits.

- Lower limit of the array = lower limit of the loop = constant variable 1
- Upper limit of the array = upper limit of the loop = constant variable 2

Note also the following topic of the programming conventions:

- [Declare limits of a loop as a constant](#) [► 1092].

General program elements for the following samples:

```

TYPE ST_Object :
STRUCT
  sName      : STRING;
  nID       : UINT;
END_STRUCT
END_TYPE

```

Negative sample:

```

VAR
  aObjects   : ARRAY[1..10] OF ST_Object;
END_VAR

```

Positive sample:

```

VAR CONSTANT
  cMin      : UINT := 1;
  cMax      : UINT := 10;
END_VAR
VAR
  aObjects  : ARRAY[cMin..cMax] OF ST_Object;
END_VAR

```

Array lower limit of 1

For arrays in the IEC 61131 languages, both the upper and lower limits can be defined explicitly. A constant variable with the value 1 should be used as lower limit, so that the number of elements in the array results as upper limit. The upper limit should also be defined by a constant variable.

This is contrary to the lower limit of 0 usually implied in high-level languages. Nevertheless, it is recommended to use a lower limit of 1 to allow a consistent and easy handling of arrays.

Depending on the specific application, other limit values may also be useful. Within a function block, however, it is essential that the method of use is consistent.

Positive sample:

```

VAR CONSTANT
  cMin      : UDINT := 1;
  cMax      : UDINT := 10;
  cMaxObjects : UDINT := 10;
END_VAR
VAR
  nIndex    : UDINT;
  aSample   : ARRAY[cMin..cMax] OF REAL;
  aObjects  : ARRAY[cMin..cMaxObjects] OF ST_Object;
END_VAR
FOR nIndex := cMin TO cMax DO
  aSample[nIndex] := 123.456;
END_FOR

```

18.3.5.4 Pointers, references, interfaces**Topics:**

1. [Temporary existence of pointers/references/interfaces to temporarily existing objects \[► 1118\]](#) [++]
2. [Reset pointer/references every cycle \[► 1119\]](#) [++]
3. [Check pointers/references/interfaces before each use \[► 1119\]](#) [++]
4. [Prefer references over pointers \[► 1120\]](#) [++]

Temporary existence of pointers/references/interfaces to temporarily existing objects

Pointers, references or interfaces that refer to temporarily existing objects should only exist as long as the objects themselves.

objects exist temporarily, for example, when instantiated in [methods \[► 90\]](#) or [functions \[► 81\]](#) or as [VAR TEMP \[► 687\]](#).

Static Analysis:

Check with the help of Static Analysis rule:

- [SA0021: Transporting the address of a temporary variable](#)

Negative sample:

```
FUNCTION_BLOCK FB_Sample_neg
VAR
    pPointerToSample : POINTER TO ST_Sample; // Sample pointer to a structure
END_VAR

METHOD PUBLIC nccl_TestMethod
VAR
    stSample          : ST_Sample;
END_VAR
-----
pPointerToSample := ADR(stSample);
// NON COMPLIANT: stSample is declared in an inner scope. So pPointerToSample points to an invalid object outside of the method.
```

Positive sample:

```
FUNCTION_BLOCK FB_Sample_pos
VAR
END_VAR

METHOD PUBLIC nccl_TestMethod
VAR
    stSample          : ST_Sample;
    pPointerToSample : POINTER TO ST_Sample; // COMPLIANT
END_VAR
-----
pPointerToSample := ADR(stSample);
```

Reset pointer/references every cycle

You should reset pointers and references every cycle.

An online change can change the addresses of statically declared variables and objects. Therefore, pointers and references pointing to such variables should be reset every cycle. For this purpose, the address of the variable can be queried again using [ADR\(\) operator \[► 700\]](#).

Pointers to dynamically created objects ([_NEW \[► 732\]](#)) may not be reset every cycle, but must remain until the object is explicitly released ([_DELETE \[► 734\]](#)).

If pointers/references are requested at the input of a method, the mandatory assignment of all method input parameters causes the pointer/references to be reset by the caller.

Negative sample:

```
FUNCTION_BLOCK FB_Sample_neg
VAR
    aBuffer : ARRAY[cMin..cMax] OF BYTE; // Sample buffer
    pBuffer : POINTER TO BYTE := ADR(aBuffer); // Sample pointer to buffer
// NON COMPLIANT: the memory address of aBuffer could change during an online change. So pBuffer could become invalid.
END_VAR
```

Positive sample:

```
FUNCTION_BLOCK FB_Sample_pos
VAR
    aBuffer : ARRAY[cMin..cMax] OF BYTE; // Sample buffer
    pBuffer : POINTER TO BYTE; // Sample pointer to buffer
END_VAR
-----
pBuffer := ADR(aBuffer); // COMPLIANT: pointer is updated before usage in implementation
```

Check pointers/references/interfaces before each use

Pointers, references and interfaces should be checked for validity before each use. For pointers and interfaces this check is done by querying for "not equal to 0" ($\neq 0$), for references the operator [_ISVALIDREF \[► 735\]\(\)](#) is used.

For special checks of a pointer, the function `F_CheckMemoryArea` of the `Tc2_System` library is available for exceptional situations, which can be used to query the memory area referenced by a pointer.

Within the condition that checks the validity of the pointer, it should not be used at the same time. If you want to implement this, you must use the operator AND_THEN. See sample 2.

Static Analysis:

Verify using the following Static Analysis rules:

- [SA0039: Possible null pointer dereferences](#)
- [SA0046: Possible use of not initialized interfaces](#)
- [SA0145: Possible use of not initialized references](#)

Thematically recommended Static Analysis rules:

- [SA0124: Dereference access in initializations](#)
- [SA0125: References in initializations](#)

General program elements for the following samples:

Function block FB_Sample implements the interface I_Sample:

```
FUNCTION_BLOCK FB_Sample IMPLEMENTS I_Sample
```

Sample program:

```
PROGRAM Sample
VAR
    pSample   : POINTER TO FB_Sample;
    refSample : REFERENCE TO FB_Sample;
    ipSample  : I_Sample;
END_VAR
```

Negative sample:

```
pSample^.DoSomething();
refSample.DoSomething();
ipSample.DoSomething();
```

Positive sample 1:

```
IF pSample <> 0 THEN
    pSample^.DoSomething();
END_IF

IF __ISVALIDREF(refSample) THEN
    refSample.DoSomething();
END_IF

IF ipSample <> 0 THEN
    ipSample.DoSomething();
END_IF
```

Positive sample 2:

```
IF ipBuffer <> 0 THEN
    IF ipBuffer.bAvailable THEN
        ipBuffer.Clear();
    END_IF
END_IF

IF ipBuffer <> 0 AND_THEN ipBuffer.bAvailable THEN
    ipBuffer.Clear();
END_IF
```

Prefer references over pointers

Prefer references to pointers if possible, since references are more type-safe compared to pointers.

However, in special cases, the use of pointers is required, for example, when pointer arithmetic is needed. Likewise, buffers of arbitrary size are readily passed to a function. Pointers are also used for this purpose.

Static Analysis:

Helpful rules for pointer use that may be required:

- [SA0017: Non-regular assignments](#)
- [SA0019: Implicit pointer conversions](#)
- [SA0061: Unusual operation on pointer](#)
- [SA0064: Addition of pointer](#)
- [SA0065: Incorrect pointer addition to base size](#)

See also:

- [POINTER \[► 767\]](#)
- [REFERENCE \[► 770\]](#)
- [Object Interface \[► 102\]](#)

18.3.5.5 Allocated variables**Topics:**

1. [Do not use direct addressing \[► 1121\] \[++\]](#)
2. [Avoid multiple write accesses to outputs \[► 1121\] \[++\]](#)
3. [Avoid overlapping memory areas of address variables \[► 1122\] \[+\]](#)

Do not use direct addressing

Direct addressing should not be used for allocated variables. Instead of direct addressing, the use of automatic addressing using the * placeholder is recommended. If the placeholders * (%I*, %Q* or %M*) are used, TwinCAT automatically performs flexible and optimized addressing.

There should also be no direct address accesses in the implementation part.

Static Analysis:

Verify using the following Static Analysis rules:

- [SA0047: Access to direct addresses](#)
- [SA0048: AT declarations on direct addresses](#)

Thematically recommended Static Analysis rule:

- [SA0005: Invalid addresses and data types](#)

Negative sample:

```
VAR
    bInputSignal AT%IX4.0 : BOOL;
    bVar          : BOOL;
END_VAR
bVar := %IX0.0;
```

Positive sample:

```
VAR
    bInputSignal AT%I* : BOOL;
END_VAR
```

Avoid multiple write accesses to outputs

Multiple write access to outputs should be avoided. Multiple write access to outputs occurs if outputs are written at more than one point in the program.

Exception to this rule: assignments in different branches of an IF or CASE statement.

Static Analysis:

Check with the help of Static Analysis rule:

- [SA0004: Multiple write access on output](#)

The rule [SA0004](#) is also included in the license-free variant [Static Analysis Light](#) [► 150].

Negative sample:

```
PROGRAM Sample_neg
VAR
  bErrorLed      AT%Q*   : BOOL;      // Machine's error LED
  bErrorDrill    : BOOL;      // Drill error status
  bErrorTransport : BOOL;      // Transport error status
  nTestCounter   : WORD;      // Simple counter
END_VAR

-----
bErrorLed := bErrorDrill OR bErrorTransport; // NON COMPLIANT: bErrorLed will be written more than
once: See action CounterInc
CounterInc();
-----

(* --- Method: CounterInc ---
   This method increments the counter and checks if nCounterValue is equal to 0 *)
bErrorLed := (nTestCounter = 0); // NON COMPLIANT: bErrorLed will be written more than
once: See program MAIN
nTestCounter := nTestCounter + 1;
```

Positive sample:

```
PROGRAM Sample_pos
VAR
  bErrorLed      AT%Q*   : BOOL;      // Machine's error LED
  bErrorDrill    : BOOL;      // Drill error status
  bErrorTransport : BOOL;      // Transport error status
  bErrorCounter  : BOOL;      // Test counter error status
  nTestCounter   : WORD;      // Simple counter
END_VAR

-----
CounterInc();
  bErrorLed := bErrorDrill // COMPLIANT
  OR bErrorTransport
  OR bErrorCounter;
-----

(* --- Method: CounterInc ---
   This method increments the counter and checks if nCounterValue is equal to 0 *)
bErrorCounter := (nTestCounter = 0); // COMPLIANT
nTestCounter := nTestCounter + 1;
```

Avoid overlapping memory areas of address variables

Overlapping memory areas of address variables should be avoided. Overlapping memory areas occur if the same storage space is used by several variables.

If an overlapping memory area is required for programming reasons, [UNION](#) [► 784]s can be used for stylistic purposes. Overlapping memory areas in other variables, e.g. address variables, should be avoided.

Static Analysis:

Check with the help of Static Analysis rule:

- [SA0028: Overlapping memory areas](#)

The rule [SA0028](#) is also included in the license-free variant [Static Analysis Light](#) [► 150].

Negative sample:

```
PROGRAM Sample_neg
VAR
  nTestWord      AT%MW2   : WORD;      // Test WORD
  nTestLowByte   AT%MB4   : BYTE;      // NON COMPLIANT: overlaps with nTestWord
  nTestHighByte  AT%MB5   : BYTE;      // NON COMPLIANT: overlaps with nTestWord
END_VAR

-----
nTestLowByte := 0; // NON COMPLIANT: writes nTestWord = 16#xx00
nTestHighByte := 0; // NON COMPLIANT: writes nTestWord = 16#00xx
nTestWord := 16#C0FF; // Writes nTestLowByte and nTestHighByte too
```

Positive sample:

```
// Structure for the positive sample
TYPE ST_2_BYTES :
STRUCT
  nLow      : BYTE;      // Low byte
  nHigh     : BYTE;      // High byte
END_STRUCT
END_TYPE

// Union for the positive sample
TYPE U_BYTE_WORD :
UNION
  stLowHigh : ST_2_BYTES; // Struct with two bytes
  nValue    : WORD;       // Test WORD to be united with the two-byte-struct
END_UNION
END_TYPE

PROGRAM Sample_pos
VAR
  uTestWordToBytes AT%MW2 : U_BYTE_WORD; // Test union
  nTestLowByte      : BYTE; // COMPLIANT: not addressed
  nTestHighByte     : BYTE; // COMPLIANT: not addressed
END_VAR

-----
uTestWordToBytes.nValue := 16#C0FF; // A test value
nTestLowByte           := uTestWordToBytes.stLowHigh.nLow; // Will be 16#FF
nTestHighByte          := uTestWordToBytes.stLowHigh.nHigh; // Will be 16#C0
```

See also:

- [AT declaration \[► 69\]](#)
- [Addresses \[► 754\]](#)

18.3.5.6 Global variables**Topics:**

1. [Use attribute 'qualified_only' for GVL \[► 1123\] \[+\]](#)
2. [Use global variables wisely \[► 1123\] \[+\]](#)

Use attribute 'qualified_only' for GVL

When defining a [global variable list \[► 72\]](#) or a [parameter list \[► 73\]](#) use the attribute `{attribute 'qualified_only'}` [\[► 822\]](#), forcing the use of the GVL namespace when using the variables. By using the namespace (e.g.: `GVL_Ctrl.bPaintingActive`) the global scope of the variable becomes clear.

Positive sample:

Global variable list "GVL_Ctrl":

```
{attribute 'qualified_only'}
VAR_GLOBAL
  bPaintingActive : BOOL;
END_VAR
```

Use global variables wisely

To prevent concurrent access and to support data encapsulation, avoid global instantiations if possible.

Likewise, you should avoid using existing global variables within function blocks if possible. You assign necessary data via input parameters.

See also:

- [Multiple tasks \[► 1127\]](#)

18.3.5.7 Strings

Topics:

1. [Identifier for size and length specifications \[► 1124\] \[++\]](#)
2. [Recommended default size \[► 1124\] \[+\]](#)
3. [Transfer of large strings \[► 1124\] \[+\]](#)
4. [Processing of large strings \[► 1125\] \[+\]](#)

Identifier for size and length specifications

Refer to the size of a string variable in bytes as "Size".

Refer the current length of a string in characters as "Len".

Positive sample:

```

VAR
    sText      : T_MaxString;
    nTextSize  : UDINT;
    nTextLen   : UDINT;
END_VAR
-----
nTextSize := SIZEOF(sText);
nTextLen  := LEN(sText);

```

Recommended default size

The recommended default size of a string variable is T_MaxString (alias of STRING(255)). This applies to the local declaration of the string as well as to the declaration of the parameter in a function/method.

If another size is fixed for the use case, this should be used as the size limit (example: T_AmsNetId).

The size of T_MaxString is defined as maximum for functions like LEN() and CONCAT() and leads to acceptable performance of regular parameter assignments as "call by value".

Static Analysis:

Thematically recommended Static Analysis rule:

- [SA0026: Possible truncated strings](#)

Positive sample:

```

FUNCTION F_Sample : BOOL
VAR_INPUT
    sParam1 : T_MaxString;
    sParam2 : T_AmsNetId;
END_VAR
PROGRAM MAIN
VAR
    sLocal1 : T_MaxString;
    sLocal2 : T_AmsNetId;
    bReturn : BOOL;
END_VAR
bReturn := F_Sample(sParam1 := sLocal1,
                   sParam2 := sLocal2);

```

Transfer of large strings

If a larger string than T_MaxString is required, the transfer should be used as "call by reference". This can be achieved using either [VAR IN OUT CONSTANT \[► 684\]](#) for read-only access or POINTER TO STRING.

An input parameter as REFERENCE TO STRING(1023) is only suitable to a limited extent, because this would always require a fixed size of the assigned variable. If this restriction is acceptable or desired, prefer this variant.

Positive sample:


```

FUNCTION F_Sample : BOOL
VAR_IN_OUT CONSTANT
    sLonger1      : STRING;
END_VAR
VAR_INPUT
    pLonger2      : POINTER TO STRING;
    nLonger2Size  : UDINT;
END_VAR

PROGRAM MAIN
VAR
    sLocal        : STRING(1023);
    bReturn       : BOOL;
END_VAR

bReturn := F_Sample(sLonger1      := sLocal,
                   pLonger2      := ADR(sLocal),
                   nLonger2Size := SIZEOF(sLocal));

```

Processing of large strings

When using T_MaxString or smaller string variables, you can use string functions like CONCAT() or LEN() (Tc2_Standard).

For larger string variables you have to use functions like CONCAT2() or LEN2() (Tc2_Uilities).

18.3.6 Runtime behavior

18.3.6.1 General

Topics:

1. [Intercept division by zero. \[► 1125\] \[++\]](#)

Intercept division by zero

To prevent runtime errors, a non-zero query should always be made before a division.

Static Analysis:

Check with the help of Static Analysis rule:

- [SA0040: Possible division by zero](#)

General program elements for the following samples:

```

FUNCTION F_Sample
VAR_INPUT
    fDividend : LREAL;
    fDivisor  : LREAL;
END_VAR
VAR
    fResult   : LREAL;
END_VAR

```

Negative sample:

```
fResult := fDividend / fDivisor;
```

Positive sample:

```

IF fDivisor <> 0 THEN
    fResult := fDividend / fDivisor;
END_IF

```

18.3.6.2 Dynamic memory

Topics:

1. [Perform dynamic memory allocations carefully \[► 1126\] \[++\]](#)

i PLC_Library Tc3_DynamicMemory

The use of dynamic memory can be simplified with the help of the [PLC library Tc3_DynamicMemory](#).

Perform dynamic memory allocations carefully

Dynamic memory allocations using `__NEW [► 732]()` should be done with care. The reason is that they change the memory requirement and array limits at runtime, for example. These changes should always be taken into account, in order to prevent unauthorized memory access resulting in a program crash.

With dynamic memory allocation it is particularly important to observe which memory is allocated and how often. Since dynamic memory allocations may affect the runtime and perhaps the memory, they should only be carried out once and at certain times, such as during startup or after a system retrofit.

Also note that dynamically allocated memory must subsequently be released again using `__DELETE [► 734]()`.

Negative sample:

```
FUNCTION_BLOCK FB_Sample_neg
VAR
    pDynamicLRealArray : POINTER TO LREAL; // Pointer to dynamic array
    nArrayCounter      : INT;             // Counter variable
    bInit              : BOOL;           // Initialize array only once
END_VAR

(* NON COMPLIANT:
1: If the amount of new LREALs is constant (in this case 10 elements), use a static ARRAY [0..10] OF LREAL.
2: If the amount of new LREALs is dynamic, do not use magic numbers. If the amount of elements (10) is changed here, there would be no way to ensure that this number is also changed in any other code parts.
3: If an array is created dynamically and assigned to a pointer, do not use this original pointer to work on the array. The start address of the array would be overwritten. *)
IF NOT bInit THEN
    pDynamicLRealArray := __NEW(LREAL, 10); // NON COMPLIANT: see above (1, 2)
    FillDynamicArray();
    bInit              := TRUE;
END_IF
```

Method FillDynamicArray:

```
FOR nArrayCounter := 1 TO 10 DO // NON COMPLIANT: see above (2)
    pDynamicLRealArray^ := 1.25;
    pDynamicLRealArray := pDynamicLRealArray + SIZEOF(LREAL); // NON COMPLIANT: see above (3)
END_FOR
```

Positive sample:

```
FUNCTION_BLOCK FB_Sample_pos
VAR
    pDynamicLRealArray : POINTER TO LREAL; // Pointer to dynamic array
    nDynamicArrayLen   : INT;             // Dynamic array length
    nArrayCounter      : INT;             // Counter variable
    pDynamicLRealTemp  : POINTER TO LREAL; // Temp. pointer to dynamic array
    bInit              : BOOL;           // Initialize array only once
END_VAR

IF NOT bInit THEN
    nDynamicArrayLen := F_CalculateBufferLen();
    pDynamicLRealArray := __NEW(LREAL, nDynamicArrayLen); // COMPLIANT
    FillDynamicArray();
    bInit              := TRUE;
END_IF
```

Method FillDynamicArray:

```
// Do not work on the array with the pointer that points to the start address of the dynamic array.
// Use a temporary pointer for working on the array to keep the start address of the array.
pDynamicLRealTemp := pDynamicLRealArray;

// The following FOR-statement is COMPLIANT if nDynamicArrayLen is the length of array in any case.
FOR nArrayCounter := 1 TO nDynamicArrayLen DO
    pDynamicLRealTemp^ := 1.25;
    pDynamicLRealTemp := pDynamicLRealTemp + SIZEOF(LREAL); // COMPLIANT: Temporary pointer used
// to work on the array
END_FOR
```

Alternative method FillDynamicArray:

```
// The following FOR-statement is COMPLIANT if nDynamicArrayLen is the length of array in any case.
FOR nArrayCounter := 1 TO nDynamicArrayLen DO
  pDynamicLRealArray[nArrayCounter-1] := 1.25;           // COMPLIANT: Pointer is not changed
END_FOR
```

18.3.6.3 Multiple tasks

Topics:

1. [Avoid concurrent accesses to memory areas \[► 1127\] \[++\]](#)
2. [Avoid concurrent accesses to function blocks \[► 1128\] \[++\]](#)
3. [Use global variables wisely \[► 1128\] \[+\]](#)

Details on protection from concurrent access are described in the chapter [Multi-task data access synchronization in the PLC \[► 366\]](#).

Avoid concurrent accesses to memory areas

You should avoid concurrent accesses to memory areas. For example, concurrent access to memory areas occurs when variables are read and/or written by more than one task with at least one write access.

NOTICE

Inconsistent variable values

For example, if a global variable is accessed from multiple task contexts and at least one access is also write access, the variable value is very likely to become inconsistent. An inconsistent variable value usually has serious consequences on the program flow and can also cause a stop of the PLC.

Static Analysis:

Verify using the following Static Analysis rules:

- [SA0006: Write access from several tasks](#)
- [SA0103: Concurrent access on not atomic data](#)

The rule [SA0006](#) is also included in the license-free variant [Static Analysis Light \[► 150\]](#).

Negative sample:

GVL_Sample:

```
VAR_GLOBAL
  nTestOutput    AT%Q*   : WORD; // Test output to be allocated
  nGlobalCounter : UDINT;
  nLastErrorID   : UDINT;
END_VAR
```

Program Sample_neg_task_M, executed in task M:

```
PROGRAM Sample_neg_task_M
VAR
  nLocalTaskM : WORD;
  fbLocalTaskM : FB_Test;
END_VAR

GVL_Sample.nTestOutput := nLocalTaskM; // NON COMPLIANT: Task N reads variable
being written by Task M whereas the accesses are not synchronized.
GVL_Sample.nGlobalCounter := GVL_Sample.nGlobalCounter + 1; // NON COMPLIANT: Task M & Task N write
to GVL_Sample.nGlobalCounter
GVL_Sample.nLastErrorID := fbLocalTaskM.nErrorID; // NON COMPLIANT: Task M & Task N write
to GVL_Sample.nLastErrorID
```

Program Sample_neg_task_N, executed in task N:

```
PROGRAM Sample_neg_task_N
VAR
  nLocalTaskN : DWORD;
  fbLocalTaskN : FB_Test;
END_VAR
```

```
nLocalTaskN      := GVL_Sample.nTestOutput;           // NON COMPLIANT: Task N reads variable
                // being written by Task M whereas the accesses are not synchronized.
GVL_Sample.nGlobalCounter := 0;                       // NON COMPLIANT: Task M & Task N write
                to GVL_Sample.nGlobalCounter
GVL_Sample.nLastErrorID := fbLocalTaskN.nErrorID;     // NON COMPLIANT: Task M & Task N write
                to GVL_Sample.nLastErrorID
```

Positive sample:**GVL_Sample:**

```
VAR_GLOBAL
  nTestOutput   AT%Q* : WORD; // Test output to be allocated
  nGlobalCounter : UDINT;
  nLastErrorID_TaskM : UDINT; // only used in Task M
  nLastErrorID_TaskN : UDINT; // only used in Task N
END_VAR
```

Program Sample_pos_task_M, executed in task M:

```
PROGRAM Sample_pos_task_M
VAR
  nLocalTaskM : WORD;
  fbLocalTaskM : FB_Test;
END_VAR

GVL_Sample.nLastErrorID_TaskM := fbLocalTaskM.nErrorID; // writes data that is declared for task M
and only used in task M
IF (* special condition to synchronize access - see InfoSys chapter 'multitask data access' *) THEN
  GVL_Sample.nTestOutput := nLocalTaskM;
  GVL_Sample.nGlobalCounter := GVL_Sample.nGlobalCounter + 1;
END_IF
```

Program Sample_pos_task_N, executed in task N:

```
PROGRAM Sample_pos_task_N
VAR
  nLocalTaskN : DWORD;
  fbLocalTaskN : FB_Test;
END_VAR

GVL_Sample.nLastErrorID_TaskN := fbLocalTaskN.nErrorID; // writes data that is declared for task N
and only used in task N
IF (* special condition to synchronize access - see InfoSys chapter 'multitask data access' *) THEN
  nLocalTaskN := GVL_Sample.nTestOutput;
  GVL_Sample.nGlobalCounter := 0;
END_IF
```

Avoid concurrent accesses to function blocks

You should avoid concurrent accesses to function blocks.

Access from several task contexts to one function block is not allowed. This concerns both the call of the body and of methods or properties. The internal process is very likely to result in inconsistent variable values. The effects are again serious consequences on the program sequence or an immediate stop of the PLC.

In a few exceptions, the definition of a function block allows access from several task contexts. If a function block is designed and developed for this use case, this is explicitly documented. In all other cases, global instantiation and use from several task contexts is not allowed.

Use global variables wisely

To prevent concurrent access and to support data encapsulation, avoid global instantiations if possible.

Likewise, you should avoid using existing global variables within function blocks if possible. You assign necessary data via input parameters.

19 Samples

In the following chapter you will find some samples that may be helpful when working with the TwinCAT 3 PLC.

19.1 Basic samples

19.1.1 First steps – state machine, timer, trigger

Description	This PLC sample shows the basic syntax as well some basic instructions and functions. The project contains an enumeration-based state machine in which the state change is controlled via the following mechanisms: <ul style="list-style-type: none"> • Timer for time monitoring (TON) • Trigger for detecting a rising edge (R_TRIG)
Sample project	https://infosys.beckhoff.com/content/1033/tc3_plc_intro/Resources/7644037771/.zip
Further information	In the documentation PLC: Your first TwinCAT 3 PLC project In the documentation PLC: Programming a PLC project

19.1.2 First steps – basic PLC elements

Description	This PLC sample shows you how to use some basic PLC elements that can be used for the structuring of a PLC project. The elements used in this project are: <ul style="list-style-type: none"> • Program [▶ 86] • Function block (FB) [▶ 84] • Function [▶ 81] • Method [▶ 90] • Structure [▶ 778] • Enumeration [▶ 781] • Global variable list (GVL) [▶ 72]
Sample project	https://infosys.beckhoff.com/content/1033/tc3_plc_intro/Resources/7643973259/.zip
Further information	In the documentation PLC: Programming a PLC project

19.1.3 STRING functions

Description	This PLC sample contains a collection of samples of the basic STRING functions. The functions shown are: <ul style="list-style-type: none"> • CONCAT • DELETE • FIND • INSERT • LEFT • LEN • MID • REPLACE • RIGHT
Sample project	https://infosys.beckhoff.com/content/1033/tc3_plc_intro/Resources/7644039435/.zip
Further information	In the documentation Tc2_Standard Library: STRING functions

19.1.4 OOP basic sample

Description	This PLC sample illustrates some of the basic functions of object-oriented programming (OOP). It features the following elements/functions: <ul style="list-style-type: none"> • Function blocks (FBs) • Methods • Properties • FB inheritance/extension • Interface (ITF) implementation and use
Sample project	https://infosys.beckhoff.com/content/1033/tc3_plc_intro/Resources/7644034443/.zip
Further information	In the documentation PLC: Object-oriented programming

19.2 Extended samples

19.2.1 OOP extended sample

Description	This PLC sample contains an object-oriented program for controlling a sorting system. The application can be controlled via an integrated visualization.
Sample project	https://infosys.beckhoff.com/content/1033/tc3_plc_intro/Resources/7644036107/.zip
Further information	In the documentation PLC: Object-oriented program for controlling a sorting plant

19.2.2 Byte-Alignment

Description	This PLC sample contains some structures with implicit or explicit byte alignment. Depending on the byte alignment used and the order of the variables within the structure, padding bytes are inserted inside or at the end of the structure.
Sample project	https://infosys.beckhoff.com/content/1033/tc3_plc_intro/Resources/7643974923/.zip
Further information	In the documentation PLC: Alignment

19.2.3 Multi-task data access synchronization

Description	<p>There are various ways of synchronizing multi-task data accesses in the PLC. The following PLC samples show the following options:</p> <ul style="list-style-type: none"> • Mutex procedure (TestAndSet, FB_1ecCriticalSection) for securing critical sections • Data exchange via synchronized buffers • Data exchange via the PLC process image
Sample project	<p>https://infosys.beckhoff.com/content/1033/tc3_plc_intro/Resources/7643978251/.zip</p> <p>https://infosys.beckhoff.com/content/1033/tc3_plc_intro/Resources/7644032779/.zip</p> <p>https://infosys.beckhoff.com/content/1033/tc3_plc_intro/Resources/7643976587/.zip</p> <p>https://infosys.beckhoff.com/content/1033/tc3_plc_intro/Resources/7643979915/.zip</p>
Further information	In the documentation PLC: Multi-task data access synchronization in the PLC

19.2.4 Library documentation reStructuredText

Description	The documentation format reStructuredText can be used for a more attractive representation of the documentation of library objects in the Library Manager. This PLC sample shows the syntax of various structure and layout elements of the reStructuredText documentation options.
Sample project	https://infosys.beckhoff.com/content/1033/tc3_plc_intro/Resources/7644044171/.zip
Further information	In the documentation PLC: Extended – reStructuredText

More Information:
www.beckhoff.com/TE1000

Beckhoff Automation GmbH & Co. KG
Hülshorstweg 20
33415 Verl
Germany
Phone: +49 5246 9630
info@beckhoff.com
www.beckhoff.com

