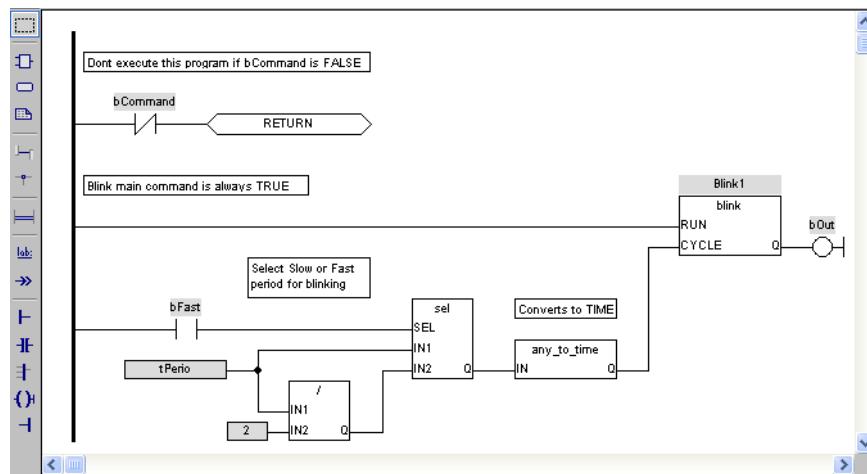
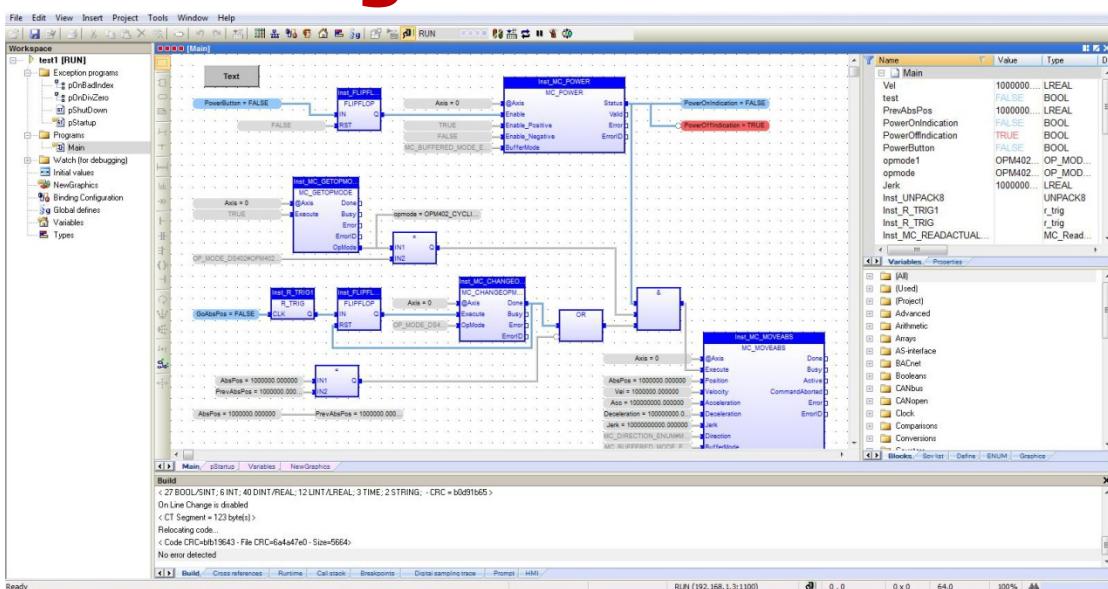


IEC 61131-3 Motion Design User Guide



February 2013 (Ver. 1.001)



www.elmomc.com

Notice

This guide is delivered subject to the following conditions and restrictions:

This guide contains proprietary Information belonging to Elmo Motion Control Ltd. Such Information is supplied solely for the purpose of assisting users of the IEC 61131-3 Motion Design User Guide servo drive in its installation.

- The text and graphics included in this manual are for the purpose of illustration and reference only. The specifications on which they are based are subject to change without notice.
- Elmo Motion Control and the Elmo Motion Control logo are trademarks of Elmo Motion Control Ltd.

Information in this document is subject to change without notice.

Document no. MAN-IEC61131-3PUG (Ver. 1.001)

Copyright © 2013

Elmo Motion Control Ltd.

All rights reserved.

Revision History

Version	Date	Details
1.001	Jan 2013	Initial document

Elmo Worldwide

Head Office

Elmo Motion Control Ltd.

60 Amal St., P.O. Box 3078, Petach Tikva 49516

Israel

Tel: +972 (3) 929-2300 • Fax: +972 (3) 929-2322 • info-il@elmomc.com

North America

Elmo Motion Control Inc.

42 Technology Way, Nashua, NH 03060

USA

Tel: +1 (603) 821-9979 • Fax: +1 (603) 821-9943 • info-us@elmomc.com

Europe

Elmo Motion Control GmbH

Hermann-Schwer-Strasse 3, 78048 VS-Villingen

Germany

Tel: +49 (0) 7721-944 7120 • Fax: +49 (0) 7721-944 7130 • info-de@elmomc.com

China

Elmo Motion Control Technology (Shanghai) Co. Ltd.

Room 1414, Huawei Plaza, No. 999 Zhongshan West Road, Shanghai (200051)

China

Tel: +86-21-32516651 • Fax: +86-21-32516652 • info-asia@elmomc.com

Asia Pacific

Elmo Motion Control APAC Ltd.

B-601 Pangyo Innovalley, 621 Sampyeong-dong, Bundang-gu, Seongnam-si, Gyeonggi-do,
South Korea (463-400)

Tel: +82-31-698-2010 • Fax: +82-31-801-8078 • info-asia@elmomc.com



Table of Contents

Chapter 1: IEC 61131-3 Motion Design Environment	1
1.1. Terminology.....	1
1.2. Setting Up a System.....	1
1.3. The Main Window and Active Icons	2
1.3.1. Working display area	5
1.3.2. Release and debug display	6
1.4. The Workspace	7
1.4.1. Project lists and Targets.....	8
1.5. Managing projects and programs.....	10
1.5.2. Defining the Project Cycle	19
1.5.3. Managing Programs.....	20
1.5.3.2. Input/Output Parameters.....	21
1.5.3.3. Description.....	24
1.6. Introduction to Creating Programs.....	25
1.6.1. Sequential Function Chart (SFC)	25
1.6.1.1. SFC Steps.....	26
1.6.1.2. SFC Transitions.....	27
1.6.1.3. SFC parallel branches.....	28
1.6.2. Jump to a SFC step.....	29
1.6.3. Actions in a SFC step.....	30
1.6.3.1. Runtime Check.....	30
1.6.3.2. Simple Boolean Actions	30
1.6.3.3. Alarms.....	30
1.6.3.4. Simple SFC actions	30
1.6.3.5. Programmed action blocks	31
1.6.3.6. Check timeout on a SFC step	31
1.6.4. SFC Transition Conditions	32
1.6.5. SFC execution at run time.....	33
1.6.6. Hierarchy of SFC programs	34
1.6.7. Controlling a SFC child program	35
1.6.8. User Defined Function Blocks programmed in SFC	36
1.6.9. Function Block Diagram (FBD)	37
1.6.9.1. LD symbols.....	38
1.6.10. Ladder Diagram (LD)	39
1.6.10.1. Use of <i>EN</i> Input and <i>ENO</i> Output for Blocks.....	39
1.6.10.2. Contacts	40
1.6.10.3. Coils.....	41
1.6.10.4. Power Rails	41
1.6.11. Structured Text (ST).....	42
1.6.12. Instruction List (IL)	43



1.6.13.	Use of ST expressions in graphic language	45
1.6.13.1.	FBD Language	45
1.6.13.2.	LD Language.....	45
1.7.	Building or Compiling the project.....	46
1.8.	Variable editor	47
1.8.1.	Creating new variables	49
1.8.2.	Variable list	50
1.8.3.	Variable attributes.....	53
1.8.4.	Initial value of a variable.....	54
1.8.5.	Variable tag and Description	55
1.9.	Editing programs	56
1.9.1.	Sequential Function Chart (SFC) Editor	57
1.9.1.1.	SFC Free Form Editor	58
1.9.1.2.	Using the SFC toolbar	58
1.9.1.3.	Moving and copying SFC charts.....	59
1.9.1.4.	Renumbering steps and transitions.....	60
1.9.1.5.	Entering step actions	60
1.9.1.6.	Entering a transition condition	62
1.9.1.7.	Entering step and transition notes	63
1.9.2.	Function Block Diagram (FBD) Editor	64
1.9.2.1.	Using the FBD toolbar.....	65
1.9.2.2.	FBD variables	67
1.9.2.3.	FBD comments.....	67
1.9.2.4.	FBD corners.....	67
1.9.2.5.	FBD network breaks.....	68
1.9.2.6.	FBD "OR" vertical rail	68
1.9.2.7.	Drawing FBD connection lines	68
1.9.2.8.	Selecting FBD variables and instances.....	69
1.9.2.9.	Viewing FBD diagrams	70
1.9.2.10.	Moving or copying FBD objects	70
1.9.2.11.	Inserting FBD objects on a line	71
1.9.2.12.	Resizing FBD objects	71
1.9.3.	Ladder Diagram (LD) Editor	73
1.9.3.1.	Using the LD toolbar	74
1.9.3.2.	Managing Rungs	75
1.9.3.3.	Comments in LD diagrams.....	76
1.9.3.4.	Viewing LD diagrams.....	76
1.9.3.5.	Moving and copying LD items.....	76
1.9.4.	Selecting function blocks	78
1.9.5.	Selecting variables and instances	79
1.10.	Project Settings.....	80
1.10.1.	Project Settings Panel.....	81
1.10.2.	Target Panel.....	83
1.10.3.	Build and Runtime Tabs	84



1.10.3.1.	Management of complex variables at runtime	86
1.10.4.	Debug window.....	87
1.10.5.	Project On Line Changes.....	88
1.11.	Definitions	90
1.11.1.	System definitions	91
1.12.	Running the application	92
1.12.1.	The control panel.....	94
1.12.1.1.	The application status.....	94
1.12.1.2.	Controlling the application, available commands	94
1.12.2.	Using the editors in test mode	96
1.12.2.1.	Locking variables.....	96
1.12.3.	The Watch window.....	97
1.12.3.1.	Managing lists.....	97
1.12.3.2.	Forcing a variable.....	97
1.12.3.3.	Dumping values	97
1.12.3.4.	Recipes.....	97
1.12.4.	Command line debugging.....	98
1.12.4.1.	Reading expressions	98
1.12.4.2.	Forcing variables.....	99
1.12.4.3.	Specifying a parent program	99
1.12.5.	Using the watch window for graphic monitoring.....	100
1.12.6.	Graphic objects.....	102
1.12.7.	Graphic objects properties	108
1.12.7.1.	Text mode	108
1.12.7.2.	Background color.....	109
1.12.7.3.	Data format.....	109
1.12.8.	Export graphics as HTML pages	113
1.12.9.	Digital sampling trace	115
1.12.9.1.	Operations	115
1.12.9.2.	Settings	115
1.12.9.3.	Start condition.....	116
1.12.9.4.	Stop condition.....	116
1.12.9.5.	Remarks	116
1.12.10.	Step by step debugging	117
1.12.10.1.	Breakpoints.....	118
1.12.10.2.	The call stack.....	118
1.12.11.	Soft oscilloscope	119
1.13.	Elmo Fieldbus configuration.....	122
1.14.	Tools	124
1.14.1.	Import / Export projects	125
1.14.2.	Monitoring applications	126
1.14.3.	The console.....	128
1.14.3.1.	Special commands	128
1.14.3.2.	Managing Programs.....	129



1.14.3.3.	Managing Program Folders.....	131
1.14.3.4.	Managing Variables	132
1.14.3.5.	Managing Data Types (structures).....	137
1.14.3.6.	Managing I/O boards.....	138
1.14.3.7.	Managing comment texts.....	139
1.15.	Resources	142
1.15.1.	String table resources.....	143
1.15.2.	Analog signals resources.....	145
1.16.	Uploading projects source code	146
1.16.1.	Save to target (download)	147
1.16.2.	Open from target (upload)	148
1.17.	Libraries	149
1.17.1.	Working with external objects	150
1.17.1.1.	Using External Objects.....	150
1.17.2.	Libraries of sub-programs and UDFBs	152
1.17.2.1.	What is a library ?	152
1.17.2.2.	Creating a library	152
1.17.2.3.	Linking a project to libraries	152
1.17.3.	Creating a User Library	154

Chapter 2: Programming languages - Reference guide 157

2.1.	Program organization units	158
2.2.	Data types.....	160
2.3.	Variables	162
2.4.	Arrays.....	165
2.5.	Constant expressions.....	166
2.6.	Conditional compiling.....	169
2.7.	Handling exceptions	170
2.8.	Variable status bits	172
2.9.	Basic Operations.....	177
2.9.1.	Assignment	178
2.9.2.	Access to bits of an integer.....	179
2.9.3.	Calling a function	179
2.9.4.	Calling a function block.....	180
2.9.5.	Calling a sub-program.....	182
2.9.6.	CASE OF ELSE END_CASE	183
2.9.7.	CountOf.....	184
2.9.8.	DEC.....	185
2.9.9.	EXIT	186
2.9.10.	FOR TO BY END_FOR.....	187
2.9.11.	IF THEN ELSE ELSIF END_IF	188
2.9.12.	INC	189
2.9.13.	Jumps JMP JMPNC JMPCN.....	190
2.9.14.	Labels	192



2.9.15.	MOVEBLOCK	194
2.9.16.	Parenthesis ()	195
2.9.17.	REPEAT UNTIL END_REPEAT	196
2.9.18.	RETURN RET RETC RETNC RETCN	197
2.9.19.	WHILE DO END WHILE	199
2.9.20.	ON	200
2.9.21.	WAIT / WAIT_TIME	201
2.10.	Boolean Operations	202
2.10.1.	FLIPFLOP	203
2.10.2.	F_TRIG	204
2.10.3.	AND ANDN &	205
2.10.4.	NOT	206
2.10.5.	OR ORN	207
2.10.6.	R	208
2.10.7.	RS	209
2.10.8.	R_TRIG	210
2.10.9.	S	211
2.10.10.	SEMA	212
2.10.11.	SR	213
2.10.12.	XOR XORN	214
2.11.	Arithmetic operations	215
2.11.1.	+ ADD	216
2.11.2.	/ DIV	217
2.11.3.	NEG -	218
2.11.4.	LIMIT	219
2.11.5.	MAX	220
2.11.6.	MIN	221
2.11.7.	MOD / MODR / MODLR	222
2.11.8.	* MUL	223
2.11.9.	ODD	224
2.11.10.	- SUB	225
2.11.11.	SetWithin	226
2.12.	Comparison operations	227
2.12.1.	CMP	228
2.12.2.	>= GE	229
2.12.3.	> GT	230
2.12.4.	= EQ	231
2.12.5.	<> NE	232
2.12.6.	<= LE	233
2.12.7.	< LT	234
2.13.	Type conversion functions	235
2.13.1.	ANY_TO_BOOL	236
2.13.2.	ANY_TO_DINT / ANY_TO_UDINT	237
2.13.3.	ANY_TO_INT / ANY_TO_UINT	238



2.13.4.	ANY_TO_LINT.....	239
2.13.5.	ANY_TO_LREAL.....	240
2.13.6.	ANY_TO_REAL.....	241
2.13.7.	ANY_TO_TIME	242
2.13.8.	ANY_TO_STRING.....	243
2.13.9.	ANY_TO_SINT	244
2.13.10.	NUM_TO_STRING.....	245
2.13.11.	BCD_TO_BIN	246
2.13.12.	BIN_TO_BCD	247
2.14.	Selectors	248
2.14.1.	MUX4.....	249
2.14.2.	MUX8.....	251
2.14.3.	SEL.....	253
2.15.	Registers	254
2.15.1.	AND_MASK	256
2.15.2.	HIBYTE.....	257
2.15.3.	LOBYTE.....	258
2.15.4.	HIWORD.....	259
2.15.5.	LOWORD.....	260
2.15.6.	MAKEDWORD	261
2.15.7.	MAKEWORD.....	262
2.15.8.	MBSHIFT	263
2.15.9.	NOT_MASK	264
2.15.10.	OR_MASK.....	265
2.15.11.	PACK8.....	266
2.15.12.	ROL.....	267
2.15.13.	ROR	268
2.15.14.	RORb / ROR_SINT / ROR_USINT / ROR_BYTE.....	269
2.15.15.	RORw / ROR_INT / ROR_UINT / ROR_WORD.....	270
2.15.16.	SETBIT	271
2.15.17.	SHL	272
2.15.18.	SHR.....	273
2.15.19.	TESTBIT	274
2.15.20.	UNPACK8	275
2.15.21.	XOR_MASK.....	276
2.16.	Counters	277
2.16.1.	CTD / CTDr	278
2.16.2.	CTU / CTUr	279
2.16.3.	CTUD / CTUDr	280
2.17.	Timers	282
2.17.1.	BLINK.....	283
2.17.2.	BLINKA	284
2.17.3.	PLS.....	285
2.17.4.	TMD	286



2.17.5.	TMU	287
2.17.6.	TOF / TOFR.....	288
2.17.7.	TON.....	289
2.17.8.	TP / TPR.....	290
2.18.	Mathematic operations.....	291
2.18.1.	ABS.....	291
2.18.2.	EXP / EXPL.....	292
2.18.3.	EXPT	293
2.18.4.	LOG	294
2.18.5.	LN	295
2.18.6.	POW ** POWL	296
2.18.7.	ScaleLin	297
2.18.8.	SQRT / SQRTL.....	298
2.18.9.	TRUNC / TRUNCL	299
2.19.	Trigonometric functions	300
2.19.1.	ACOS /ACOSL	300
2.19.2.	ASIN / ASINL.....	301
2.19.3.	ATAN / ATANL.....	302
2.19.4.	ATAN2 / ATANL2.....	303
2.19.5.	COS / COSL.....	304
2.19.6.	SIN / SINL	305
2.19.7.	TAN / TANL	306
2.19.8.	UseDegrees.....	307
2.20.	String operations	308
2.20.1.	ArrayToString / ArrayToStringU.....	309
2.20.2.	ASCII.....	310
2.20.3.	ATOH.....	311
2.20.4.	CHAR	312
2.20.5.	CONCAT	314
2.20.6.	CRC16.....	315
2.20.7.	DELETE	316
2.20.8.	FIND	317
2.20.9.	HTOA.....	318
2.20.10.	INSERT.....	319
2.20.11.	LEFT.....	320
2.20.12.	LoadString	321
2.20.13.	MID	322
2.20.14.	MLEN.....	323
2.20.15.	REPLACE.....	324
2.20.16.	RIGHT	325
2.20.17.	StringTable	326
2.20.18.	StringToArray / StringToArrayU.....	327
2.21.	Advanced operations.....	328
2.21.6.	Communication	330



2.21.8.	PTPALARM_A.....	331
2.21.9.	ALARM_M	332
2.21.10.	ApplyRecipeColumn.....	333
2.21.11.	AS-interface functions	334
2.21.12.	AVERAGE / AVERAGEL	335
2.22.	CANopen functions.....	336
2.22.1.	CanRcvMsg.....	338
2.22.2.	CanSndMsg	338
2.22.3.	CycleStop	339
2.22.4.	CurveLin	339
2.22.5.	DERIVATE	340
2.23.	DNP3 Master function blocks.....	341
2.23.1.	DNP3_CMDANLG	341
2.23.2.	DNP3_CMDBIN	342
2.23.3.	DNP3_EV POLL.....	343
2.23.4.	DNP3_FREEZCNTR	343
2.23.5.	DNP3_INTPOLL	343
2.23.6.	DNP3_READCLASS	344
2.23.7.	DNP3_READGROUP	345
2.23.8.	DNP3_READPTS	345
2.23.9.	DNP3_WRITEBIN.....	346
2.23.10.	DNP3_WRI TESTR.....	346
2.23.11.	Command and status values.....	347
2.24.	Dynamic memory allocation functions.....	349
2.24.1.	ARCREATE	350
2.24.2.	ARREAD.....	350
2.24.3.	ARWRITE	350
2.24.4.	EnableEvents	351
2.24.5.	FatalStop.....	351
2.24.6.	FIFO	352
2.25.	File Management functions.....	354
2.25.1.	F_ROPEN	355
2.25.2.	F_WOPEN.....	355
2.25.3.	F_AOPEN	355
2.25.4.	F_CLOSE	355
2.25.5.	F_EOF.....	356
2.25.6.	FA_READ	356
2.25.7.	FA_WRITE	356
2.25.8.	FM_READ	356
2.25.9.	FM_WRITE	357
2.25.10.	FB_READ	357
2.25.11.	FB_WRITE.....	357
2.25.12.	F_EXIST.....	357
2.25.13.	F_GETSIZE	358



2.25.14. F_COPY.....	358
2.25.15. F_DELETE	358
2.25.16. F_RENAME.....	358
2.25.17. GETSYSINFO	359
2.25.18. HYSTER.....	361
2.25.19. INTEGRAL.....	362
2.25.20. LIFO	363
2.25.21. LIM_ALRM.....	365
2.25.22. LogFileCSV.....	366
2.25.23. SetCsvOpt	368
2.25.24. MBMasterTCP.....	370
2.25.25. MBSlaveRTU / MBSlaveRTUex.....	372
2.25.26. MBSlaveUDP / MBSlaveUDPex	373
2.26. MQTT (MQ Telemetry Transport) Protocol Handling.....	374
2.27. MQTT Connection and Status.....	375
2.27.1. MQttConnection.....	375
2.27.2. MQttLastError.....	377
2.27.3. MQttMsgStatus	379
2.28. MQTT Messaging	380
2.28.1. MQttPublish.....	380
2.28.2. MQttSubscribe.....	381
2.28.3. MQttUnsubscribe	381
2.28.4. MQttReceivePub.....	382
2.28.5. MQttNbRec.....	383
2.29. Additional MQTT functions	384
2.29.1. MQttIECFormat.....	384
2.29.2. MQttIECParse	385
2.30. MQTT Protocol Handling (Cont.)	386
2.30.1. PID.....	386
2.30.2. Printf	390
2.30.3. RAMP	391
2.31. Real Time Clock management functions	393
2.31.1. Real Time Clock.....	394
2.31.2. DAY_TIME	396
2.31.3. DTFORMAT	397
2.31.4. DTAT	398
2.31.5. DTEVERY	400
2.31.6. Serializeln.....	401
2.31.7. SerializeOut.....	403
2.31.8. SerGetString.....	405
2.31.9. SerPutString	407
2.31.10. SERIO.....	409
2.31.11. SigID	411
2.31.12. SigPlay.....	412



2.31.13. SigScale	414
2.31.14. STACKINT	415
2.31.15. SurfLin	417
2.32. TCP/IP Management.....	418
2.32.1. TCP/IP Management Functions	419
2.33. Text buffers manipulation	424
2.33.1. Memory management / Miscellaneous	425
2.33.2. Allocation / exchange with files.....	427
2.33.3. Data exchange	429
2.33.4. Sequential reading.....	432
2.33.5. Sequential writing.....	433
2.33.6. UNICODE conversions.....	435
2.34. UDP Management	436
2.34.1. UDP Management Functions.....	437
2.35. VLID.....	439
2.36. XML writing and parsing.....	440
2.36.1. XmlManager	442
2.36.2. XmlLastError	443
2.36.3. Reading/writing documents	444
2.36.4. Exploring an XML document.....	447
2.36.5. Building an XML document.....	451
2.36.6. Exchanging variables using their name	453
2.37. T5 Registry management.....	454
2.37.1. Parameter pathnames.....	455
2.37.2. T5 Registry management functions.....	456
Chapter 3: Embedded HMI.....	457
3.5. Embedded HMI graphic objects	459
3.6. Embedded HMI Functions and Function blocks	464
3.7. Managing menus	465
Chapter 4: T5 Registry for runtime parameters	468
4.1. Designing the registry of parameters.....	468
4.2. Monitoring parameters	469
Chapter 5: Project Automation Library	470
5.1. Creating a new Project Automation script	471
5.2. Developing and testing the script.....	473
5.3. Reference.....	477
5.3.1. Project Level Services	478
5.4. Declaring Programs	483
5.4.1. paCreateProgram	484
5.4.2. paCreateSubProgram	485
5.4.3. paCreateUDFB	486



5.4.4.	paCreateSfcChildProgram.....	487
5.4.5.	paCopyProgram	487
5.4.6.	paSetProgramComment	488
5.4.7.	paEnumProgram	489
5.4.8.	paGetProgramDesc.....	490
5.4.9.	paDeleteProgram.....	491
5.4.10.	paCreateProgramFolder	491
5.4.11.	paSendProgramToFolder.....	492
5.4.12.	paEnumProgramFolder.....	493
5.4.13.	paDeleteProgramFolder	494
5.5.	Declaring data structures	495
5.5.1.	Data Structure Functions.....	495
5.6.	Declaring Variables	498
5.6.2.	paCreateVar.....	499
5.6.3.	paCreateInputParam	500
5.6.4.	paCreateOutputParam	501
5.6.5.	paSetVarDim	502
5.6.6.	paSetVarInitValue	503
5.6.7.	paSetVarComment	503
5.6.8.	paEmbedVarSymbol	504
5.6.9.	paProfileVar	504
5.6.10.	paEnumVar	505
5.6.11.	paGetVarDesc	506
5.6.12.	paDeleteVar	507
5.7.	Declaring I/O Boards.....	508
5.7.1.	I/O Board Functions.....	508
5.7.2.	paEnumIBoards.....	511
5.7.3.	paGetIBoardDesc	512
5.7.4.	paDeleteIBoard	512
5.7.5.	paDeleteAllIBoards	512
Chapter 6:	Generating documents.....	513
6.1.	Common File Services.....	513
6.1.1.	paFileOpenWrite.....	513
6.1.2.	paFileOpenWriteProgramSrc.....	514
6.1.3.	paFileOpenWriteProgramDef	514
6.1.4.	paFileClose.....	515
6.2.	Text File Writing Services	516
6.2.1.	Text File Writing Functions.....	516
6.3.	IEC Source Code File Writing Services	517
6.3.1.	ST / IL Text	517
6.3.2.	ST/IL Text Functions.....	517
6.3.3.	FBD Sequential File Writing Services	519
6.3.4.	FBD Sequential File Writing functions	520



6.3.5.	LD File Writing Services	526
6.3.6.	LD File Writing Functions.....	527
6.3.7.	SFC File Writing Services.....	534
6.4.	Network and Fieldbuses File Writing Services	541
6.4.1.	Network and Fieldbuses File Writing Functions	543
6.5.	Watch Window Documents File Writing Services	564
6.5.1.	Watch Window Documents File Writing Functions.....	565
6.6.	Resource Documents File Writing Services	568
6.6.1.	Resource Documents File Writing Functions.....	569
6.7.	Templates	572
6.7.2.	Variable Keyword Functions.....	572
6.7.3.	Templates of programs.....	573
6.7.4.	Template Variable Keyword Functions.....	573
6.8.	Script parameters	576
6.8.1.	Script Related Functions.....	576
6.9.	Miscellaneous.....	578
6.9.1.1.	paGetLanguage.....	578
6.9.1.2.	paTRACEx.....	578
Chapter 7:	Workbench COM Interface	579
7.1.1.	day_time_local	580
Chapter 8:	PID Functions by JS Automation	581
Chapter 9:	IDE Library Functions.....	583
9.1.1.	ROLb / ROL_SINT / ROL_USINT / ROL_BYTE.....	583
9.1.2.	ROLw / ROL_INT / ROL_UINT / ROL_WORD	584
9.1.3.	ApplyRecipeColumn RTK_IsBugCheck	585
9.1.4.	Terminate	586
9.1.5.	RedSwitch	586
9.1.6.	ApplyRecipeColumn RTK_OnBugCheck.....	587
9.1.7.	Stop.....	588
9.2.	Run Time Options	589
9.2.2.	Program scheduling	589
9.2.3.	SHLb / SHL_SINT / SHL_USINT / SHL_BYTE.....	590
9.2.4.	SHLw / SHL_INT / SHL_UINT / SHL_WORD	591
9.2.5.	SHRb / SHR_SINT / SHR_USINT / SHR_BYTE.....	592
9.2.6.	SHRw / SHR_INT / SHR_UINT / SHR_WORD	593
9.2.7.	SET_DAY_TIME	594



Chapter 1: IEC 61131-3 Motion Design Environment

The purpose of this programming environment is to configure and design a motion system according to the engineering requirements of specific applications, using the G-MAS motion controller.

1.1. Terminology

Term	Explanation
EAS	Elmo Application Studio
IEC	International Electrotechnical Commission
UDFBs	User Defined Function Blocks
ST	Structured Text
IL	Instruction Language
FBD	Function Block Diagram
LD	Ladder Diagram
SFC	Sequence Flow Chart (Sequential Function Chart)
POU	Program Organization Units includes Program, Function, and Function Block
MMS	Manufacturing Message Specifications

1.2. Setting Up a System

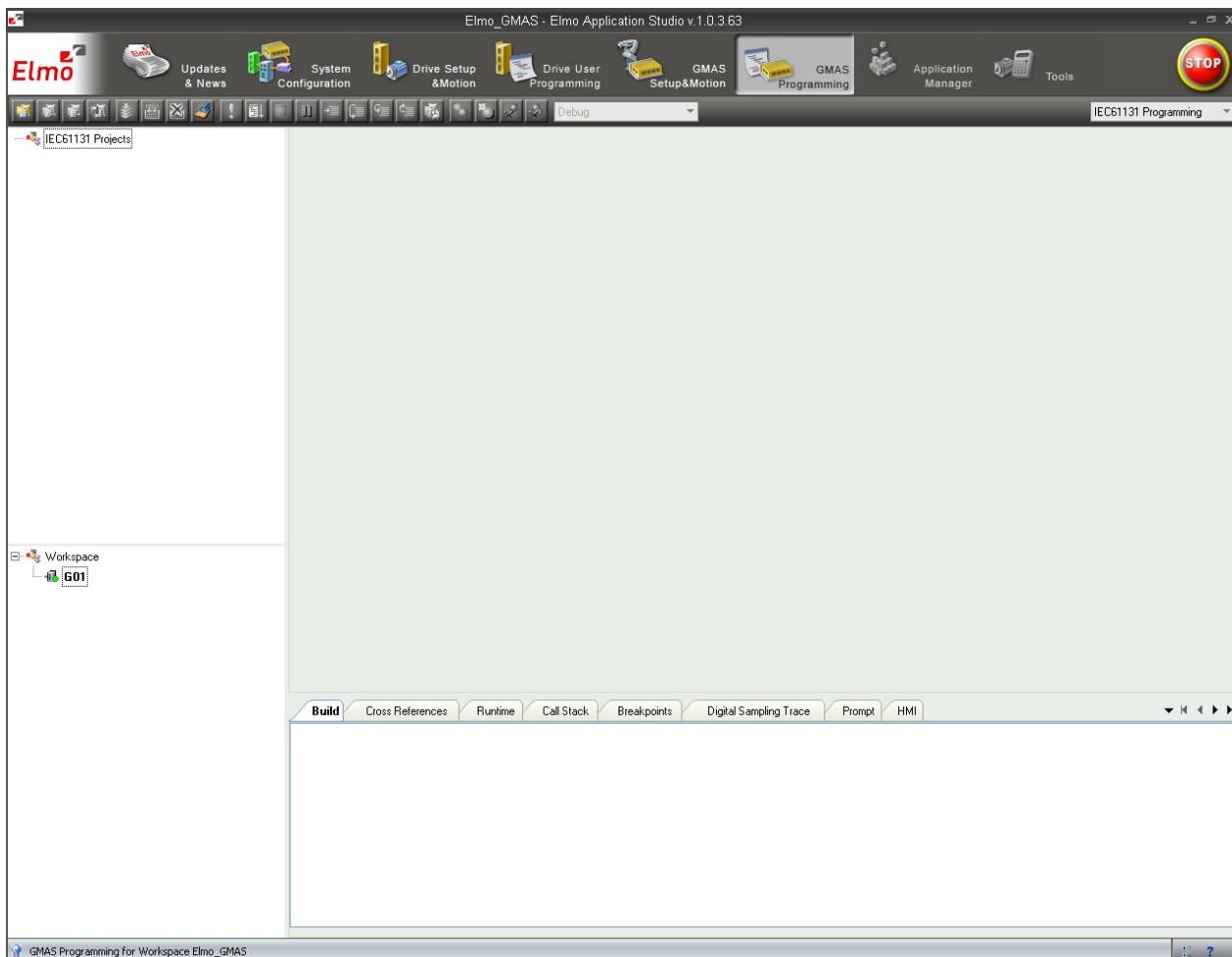
The servo-drives, motion controllers, I/Os and any other sections of the system are configured within the Elmo Application Studio (EAS) environment as detailed in the EAS User Guide.

To set up the IEC 61131-3 programming environment do the following:

1. Make sure that the hardware system is configured within the EAS application.
2. Within the EAS application, select **GMAS Programming** at the title bar.
The IEC 61131-3 programming environment appears.



1.3. The Main Window and Active Icons



The main toolbar of the main window is a control panel that groups main commands of the Workspace:

When a new or previous project is opened the following icons are available.



Icon Description



Add a new project



Opens the Import Assistant to import an XML project, open a previous version of the project, or import an ST file.



Opens the Export Assistant to export a complete project , definitions, or previous version of the project.



Saves all projects opened and attached to the IEC61131 Projects tree.



When a project is opened the following icons are active.



Icon	Description
	Compiles a project
	Builds a project
	Stop building a project
	Clears a project from the workspace window.
	Start without debugging
	Start debugging
	Stop debugging
	Pause debugging

When the project is linked to the G-MAS motion controller, and running, the following icons are available.



Icon	Description
	Single cycle.
	Step over
	Step into
	Step outside
	Project cycle

When editing the program, the following icons are available.



Icon	Description
	Add breakpoint

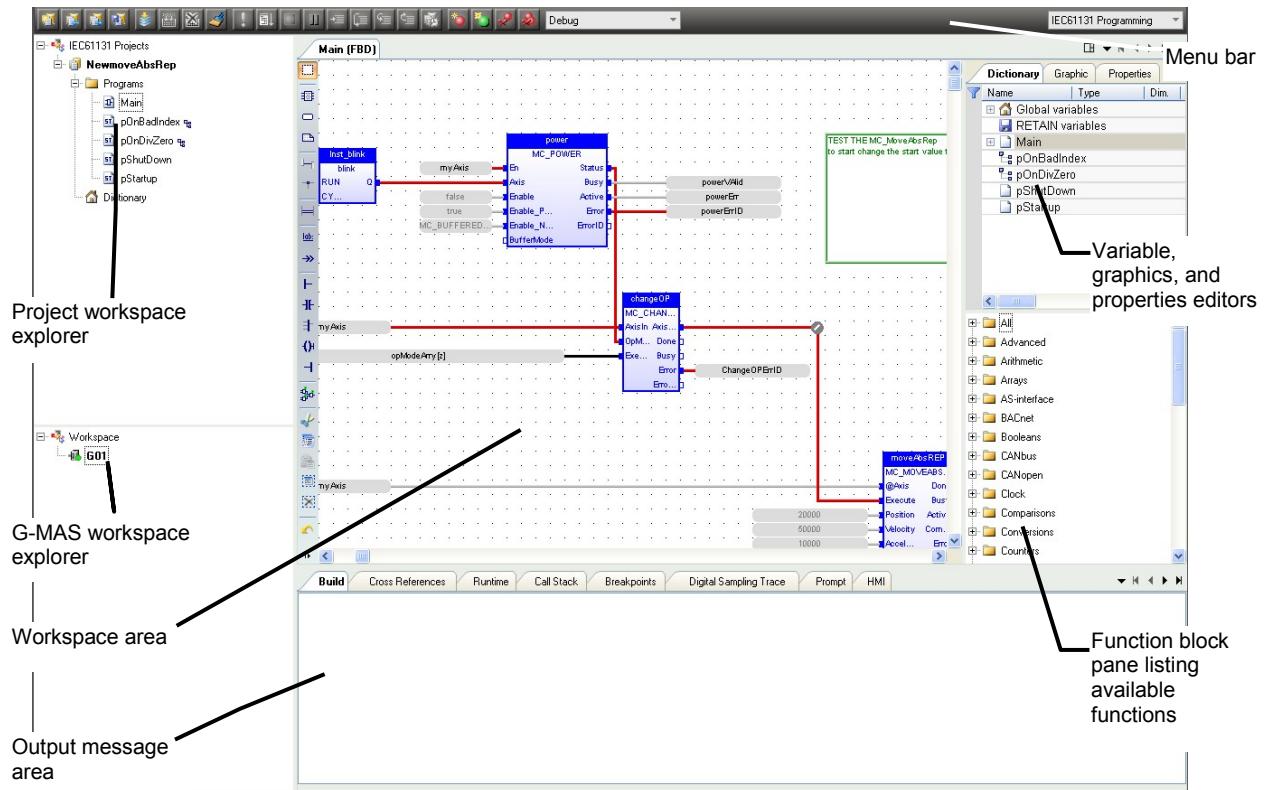


Icon	Description
	Add tracepoint
	Remove breakpoint
	Remove all breakpoints

When the mouse is placed over any icon a textbox displays the icon's purpose.

1.3.1. Working display area

The workspace displays the following:



Every fixed area can be hidden or shown using the commands of the View menu. In the middle area are open documents of the workspace. In case several documents are open together, use the tab control at the bottom of the area to select the active document.



1.3.2. Release and debug display

The main window provides two operational and two display views for the Debug or Release modes and IEC61131 Programming or Debugging modes.

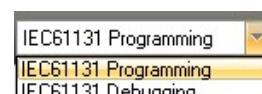


By default the operational mode is Debug and the display view is IEC61131 Programming.

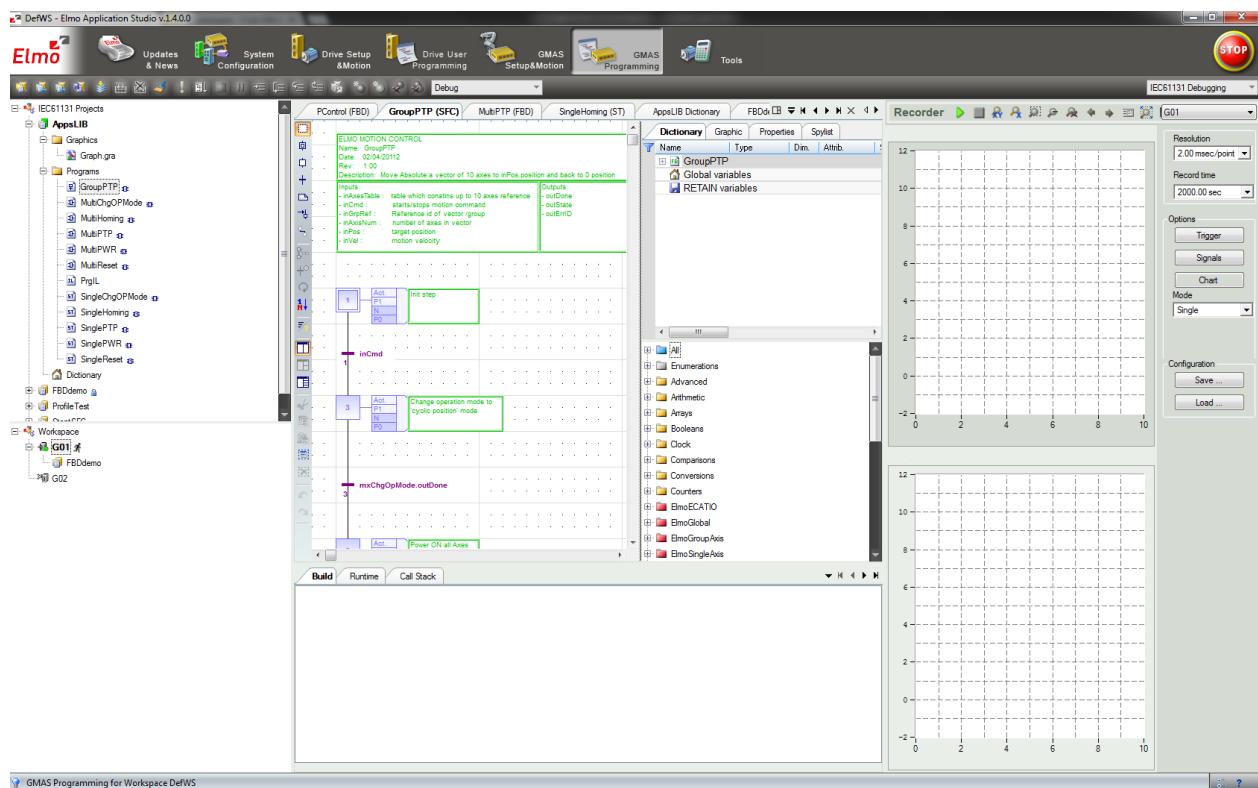
To change the operational mode select from the pulldown menu.



To change the display mode, select from the top right corner pulldown menu.

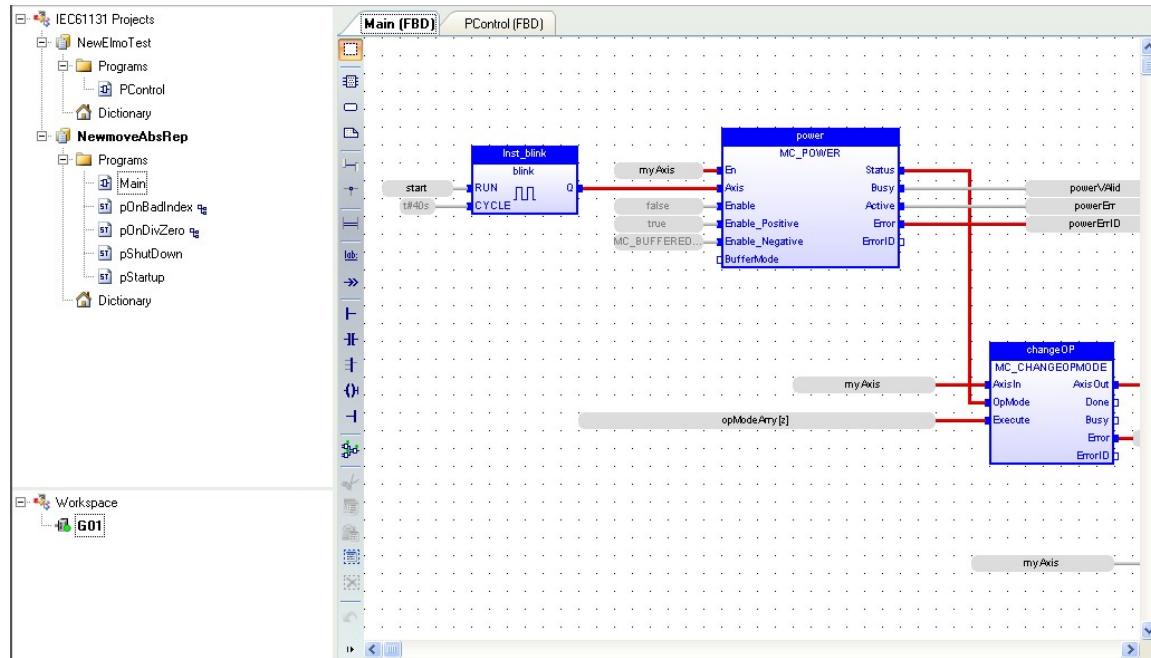


When the IEC61131 Debugging mode is selected, the view changes to the following:





1.4. The Workspace

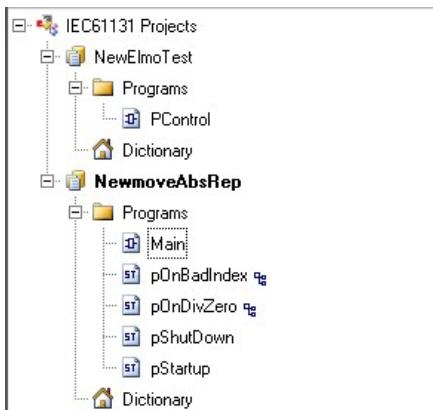


The Workbench enables editing of several projects in the same workspace. Projects of the workspace are shown in the left-hand window of the Workbench, with their contents (programs, lists...). The set of projects edited in the Workbench is the Project List in menu commands.

Practically, each project is stored as a folder on the disk in the main directory **IEC61131Projects** situated under the user **WorkSpace** directory. Any relevant Information about a project is stored with that folder. All projects under this directory can be easily opened and read into the application in EAS. Imported XML projects will also automatically create the relevant folder under the main directory. However, projects copied to subfolders under the main directory will not be seen by EAS.



1.4.1. Project lists and Targets

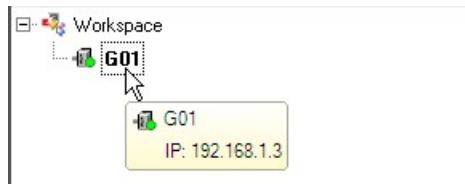


Use the icon commands listed in section 1.3.1 to manage the project lists. Select a program to be displayed in the Workspace area for editing, running or otherwise. The root project is highlighted. This now becomes the default Startup Project, the project opened when entering On Line mode. Only one project at a time can be tested. To change the Startup Project, select and highlight another project name in the Workspace window.

Below the project list is the Targets; the motion controller(s) responsible for running the projects created in the Project List. Make sure to setup and configure the motion controller from the System

Configuration , as described in the EAS User Guide.

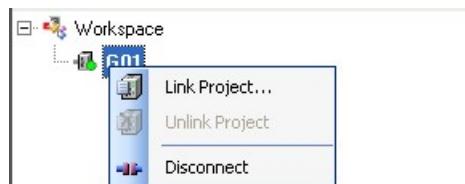
At the lower panel below the workspace explorer is Workspace Target; the G-MAS or other motion controller. Move the mouse over the motion controller to view its IP address.



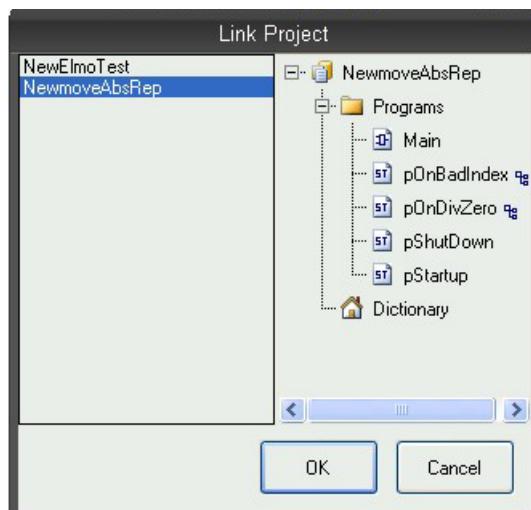
To run the program, it is necessary to link the program to Workspace Target.

To link between an open program and a Target:

1. Right-click on the Workspace Target. A menu opens.



2. From the menu, select **Link Project**. The Link Project window box opens.



3. Select the appropriate program to link to the target, and then click **OK**.
The project and its programs are then linked to the target.



Optionally, drag the Project to the Target. The Project is linked to the Target. If a previous Project is presently linked, the new project replaces the previous one.

4. When a program becomes linked to a target, the project becomes locked and displays a lock at the side of the highlighted project name.



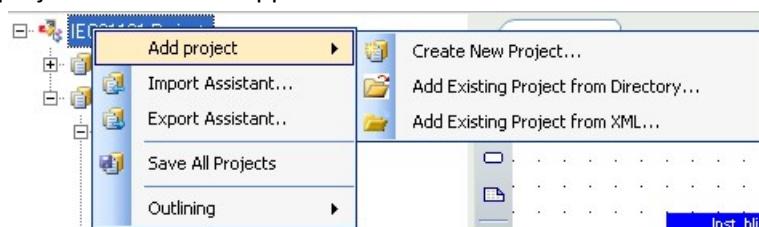
1.5. Managing projects and programs

Projects are managed from the Workspace Explorer at the left side of the Workspace main window. Here the projects and programs are managed and the User Defined Function Blocks (UDFBs) of your application, declared. The workspace displays the list of all programs and UDFBs, initially sorted by alphabetic order within a project.

1.5.1. Managing Projects

To manage projects:

1. From the main IEC61131 Projects Workspace explorer root, right-click on the root IEC61131 Projects. The projects main menu appears.



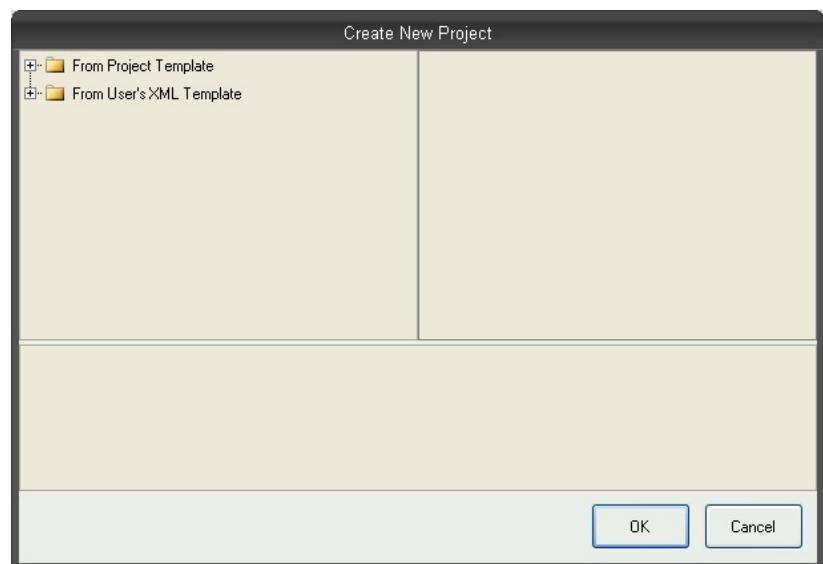
Alternative, select the appropriate icon described in step 2, from the icon bar menu.

2. Select an option from the menu, or click the icon directly:

Add Project Allows selection of three options:

Create New Project Allows creation of a new project based on the previously defined project template,

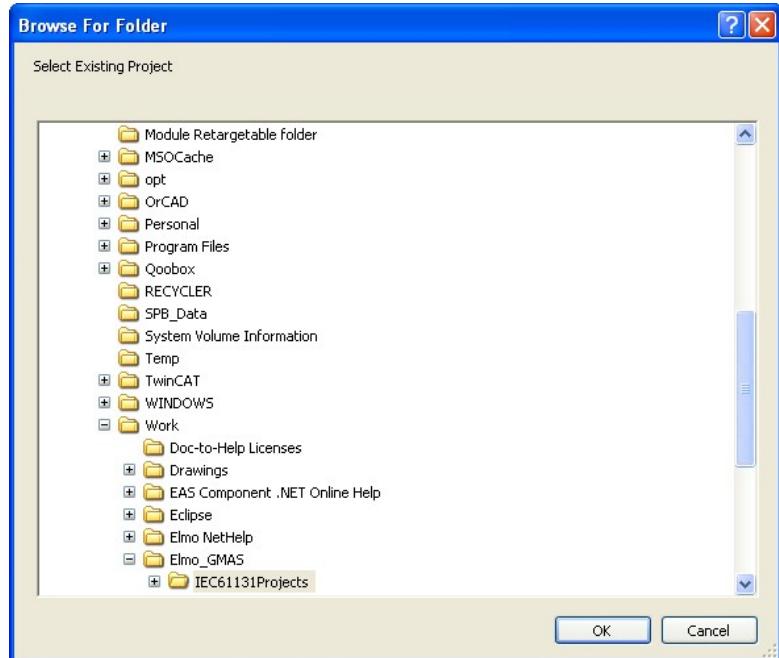
Or, double-click the icon . The Create New Project window opens.



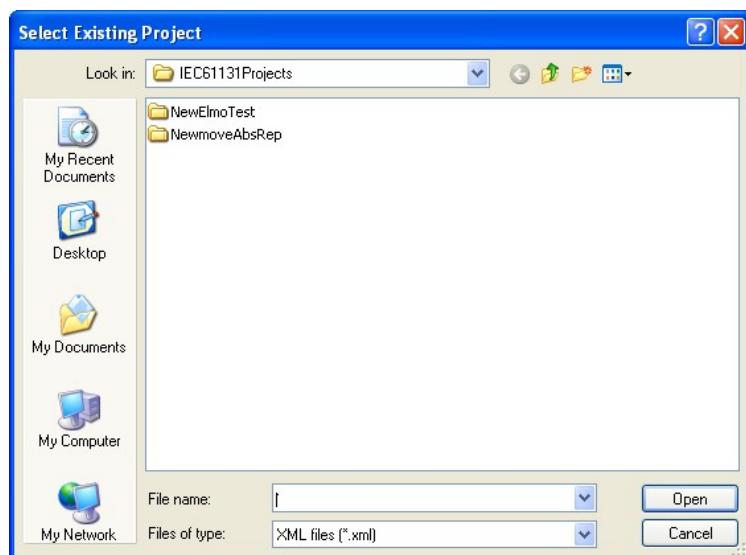


Add Existing Project from Directory Adds an existing project present on any local or network directory linked to the system,

Or, double-click the icon . The Browse for Folder window opens.

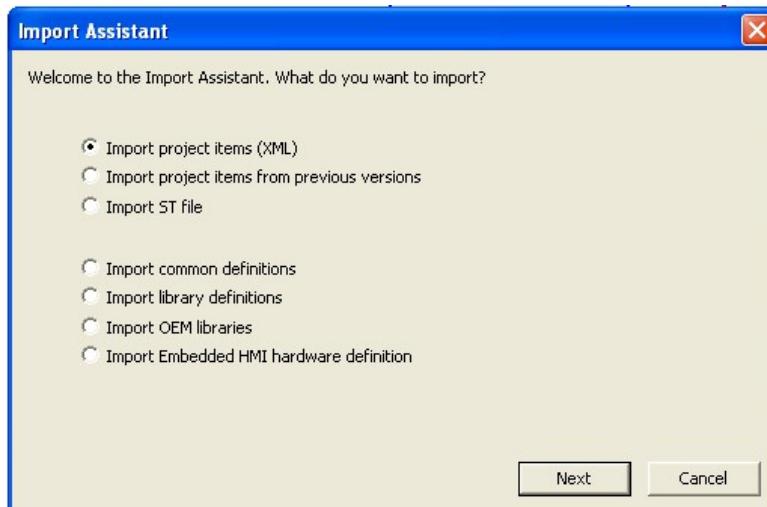


Add Existing Project from XML Adds a project from an XML file present on any local or network directory linked to the system. The Select Existing Project window opens.





Import Assistant Offers a number of options to import sections of a project, or an ST file.



The following options are not available in EAS:

- Import project items from previous versions
- Import Embedded HMI hardware definition (not available at this stage)

Export Assistant

Offers a number of options to export sections of a project, or a complete project to an XML or ST file.



The following option is not available in EAS:

- Export a project item to previous versions

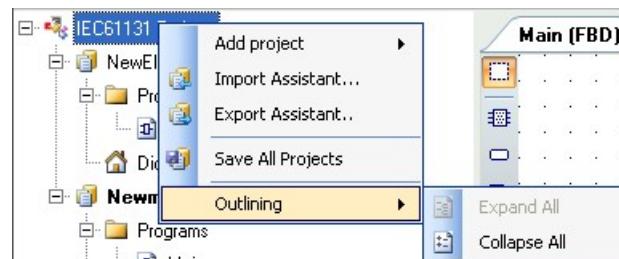
Save All Projects

Saves all projects listed attached to the main IEC61131 Projects list.



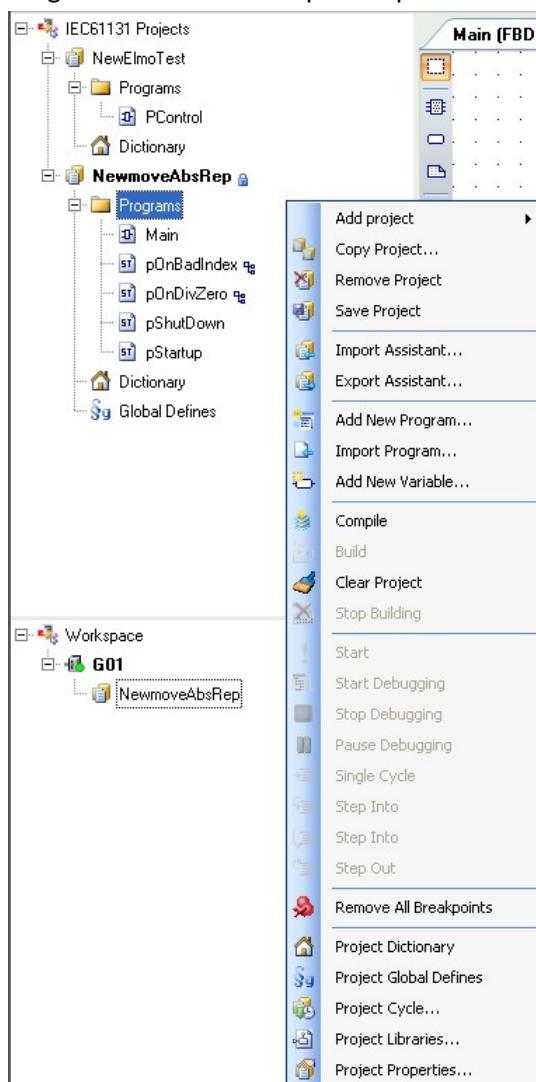
Outlining

Defines whether the Projects list is fully expanded or collapsed.



To manage a specific project:

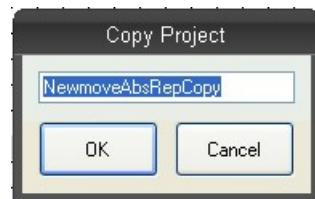
1. Highlight the project and right-click in the Workspace Explorer area. A menu opens.



2. Select an option from the menu:

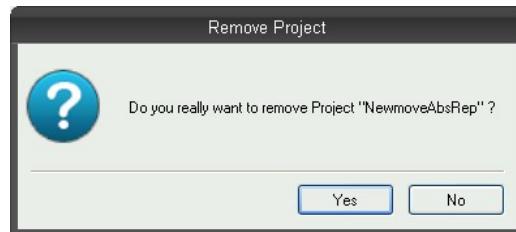
Add Project Allows selection of the three options described above for the IEC611331 Projects menu.

Copy Project Allows copying of the project or alternative renaming of the project using the same programs. Copy cannot be used for erasing an existing program. You must enter a new name for the destination of the copy.



Click **OK** to copy the new named project.

Remove Project Removes the selected project. Click **Yes** to remove the project.



Import Assistant Offers a number of options to import sections of a project, or an ST file.

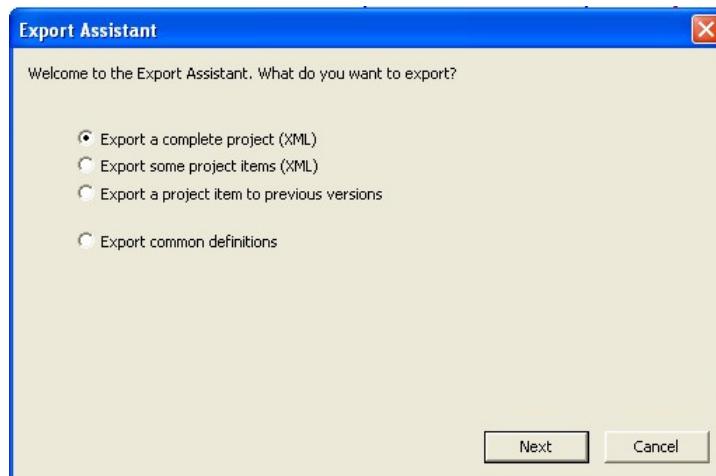


The following options are not available in EAS:

- Import project items from previous versions
- Import Embedded HMI hardware definition (not available at this stage)



Export Assistant Offers a number of options to export sections of a project, or a complete project to an XML or ST file.

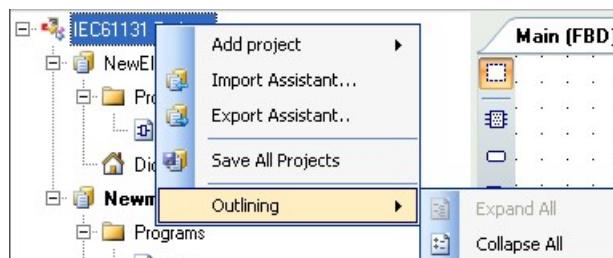


The following option is not available in EAS:

- Export a project item to previous versions

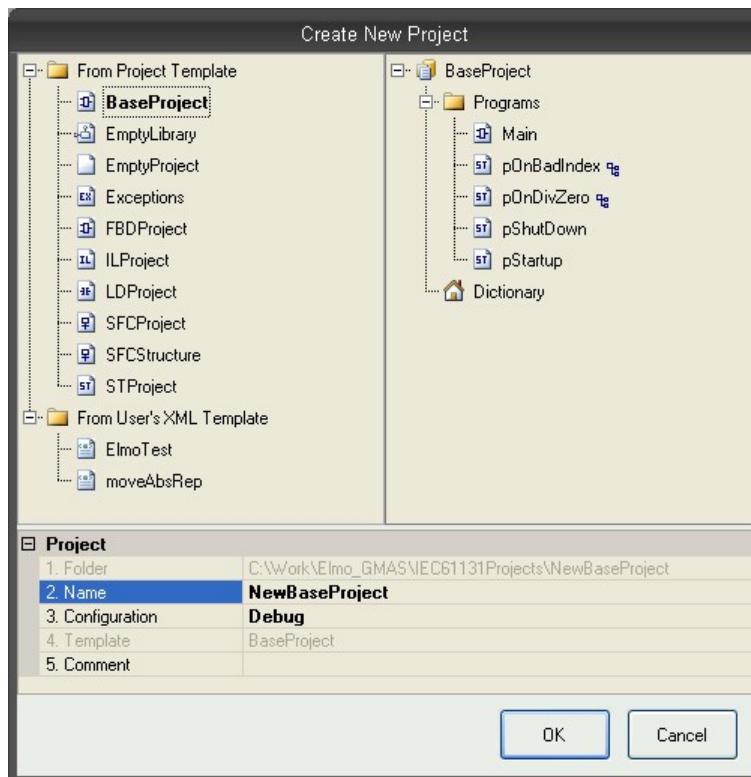
Save All Projects Saves all projects listed attached to the main IEC61131 Projects list.

Outlining Defines whether the Projects list is fully expanded or collapsed.

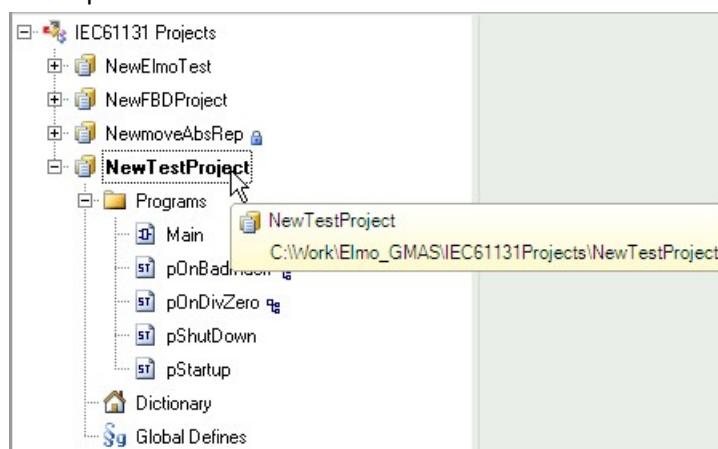


3. If in the **Create New Project** window:

- a. Open the expanding tree on the left panel to view the Project and User's XML Template options.



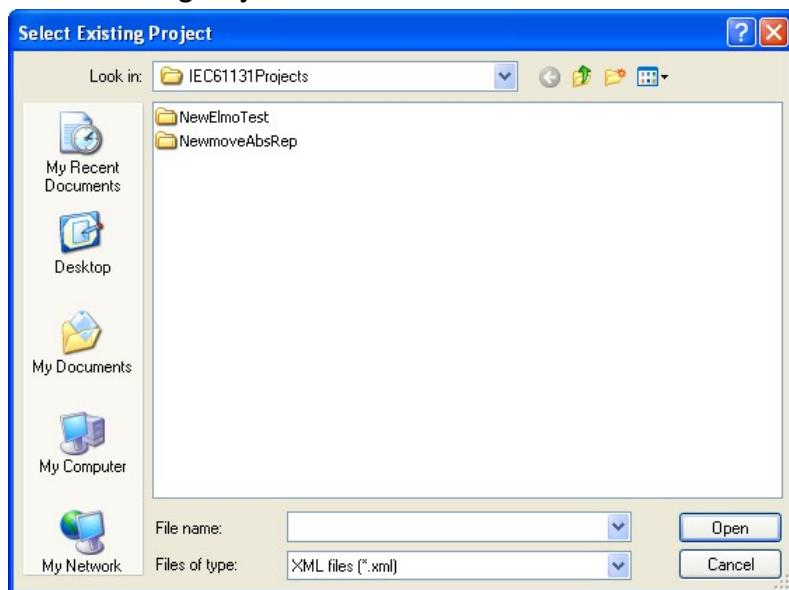
- b. Select a template option, and under the **Project** heading change the **2. Name** to a relevant name.
- c. Then click **OK** to accept the project definitions. The new Project is created and viewable in the WorkSpace area.



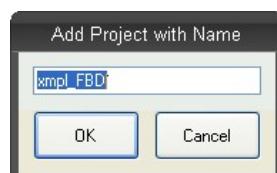
4. If necessary to import an XML project:



a. Select Add Existing Project from XML.



b. Select the existing XML file, and click **Open**. The Add Project with Name window box opens.



c. Click **OK**. The project appears in the WorkSpace.

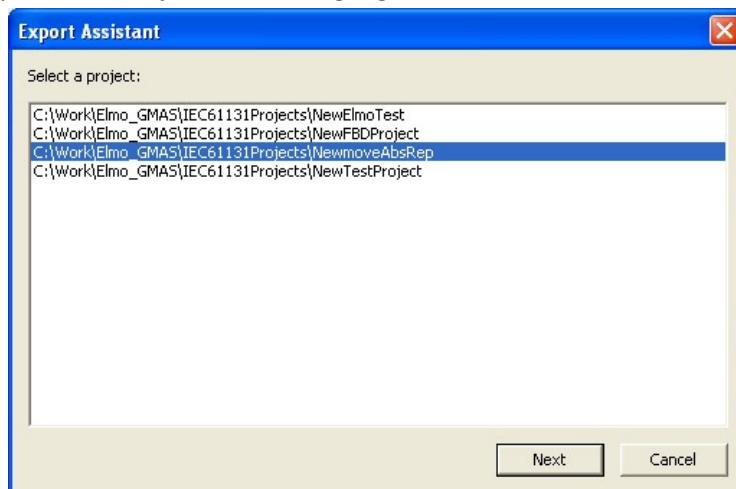
5. If in the **Export Asistant**, to export a complete project to an XML format:

a. Select the appropriate export option. Then click **Next**.

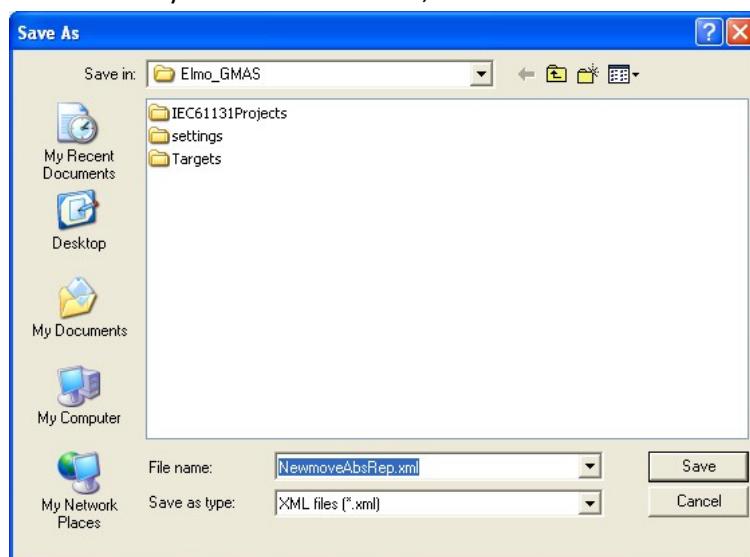




- b. Verify that the Project name is highlighted, then click **Next**.



- c. Select the directory to save the XML file, and click **Save**.

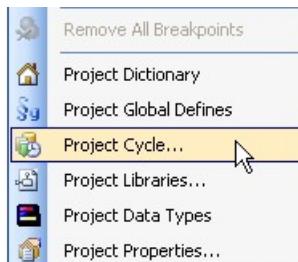




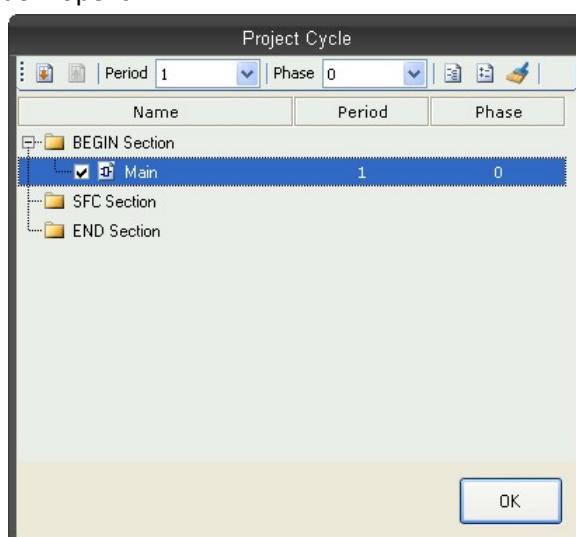
1.5.2. Defining the Project Cycle

To define the executing sequence of a project:

1. Right-click on the project and select **Project Cycle...**



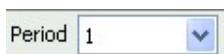
The Project Cycle window opens.



2. The Project Cycle window has the following options:

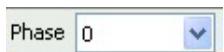


Move program down or up to change the order of the programs within the cycle.

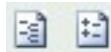


Select the cycle period to specify if the program must be called in the cycle. Unselected programs are not kept at compiling time, and are shown with a red cross icon in the workspace.

Use the *Increase/Decrease* buttons to change the scheduling of a program, enabling the defining of "slow" programs that are not called every cycle.



Selectable for periods >2, as required.



Expand or Collapse the program tree connected to a Project.



Reset all project cycles and phases to their default values.

3. Adjust the project cycle as required and then click **OK**.



1.5.3. Managing Programs

Programs are grouped in the following Program Organization Units (POUs):

- Program
- Function
- Function Blocks

For details of the POU, refer to the section 3.1 Program organization units on page 158.

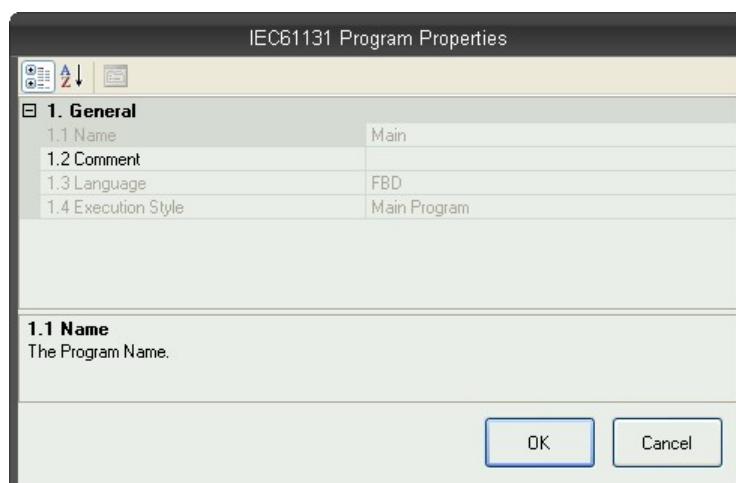
Programs have unique names. The name cannot be a reserved keyword of the programming languages and cannot have the same name as a standard or "C" function or function block. A variable should not have the same name as a declared variable. The name of a program should begin by a letter or an underscore ("_") mark, followed by letters, digits or underscore marks. It is not allowed to put two consecutive underscores within a name. Naming is case insensitive. Therefore, two names with different cases are considered as the same.

To open a program:

1. Press **Enter** or double click on a program to open it with the appropriate editor. An open program cannot be renamed or deleted.

1.5.3.1. Program properties

The Program Properties box enables you to define the properties of a program or function block (refer to the section 3.1).



When creating a new POU:

1. Select its programming language. The programming language cannot be changed later.
2. Choose an Execution Style for the POU:
 - Main program (called in the cycle)
 - Sub-program (explicitely called by other programs)
 - UDFB (User Defined Function Block)
 - Child SFC program (activated by its parent SFC program)
 - For a CHILD SFC program, you have to select its parent SFC program. This parent can be changed later.

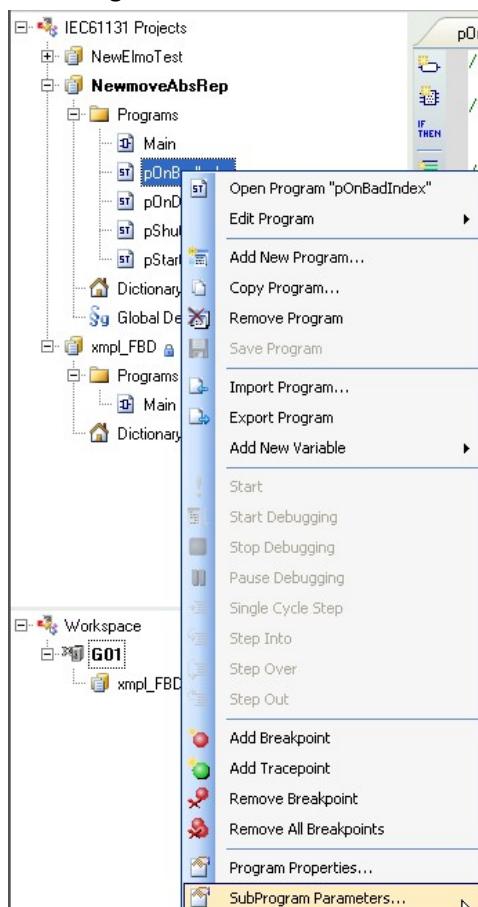


1.5.3.2. Input/Output Parameters

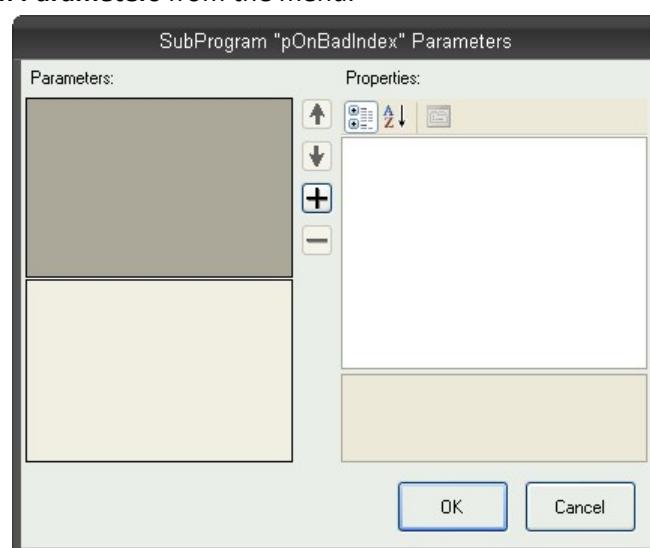
For a UDFB or sub-program, the **Program Properties** allows the user to define and arrange input and output parameters.

To edit the Program properties for a UDFB or sub-program:

1. Select UDFB or sub-program and right-click.



2. Select **SubProgram Parameters** from the menu.





3. It is now possible to enter and display two lists of parameters, for inputs and outputs. Each parameter has a name, description, Definition, and Dimension.



Move the selected parameter in the list and thus arrange the order of parameters. The order has a strong importance as it defines the calling prototype of the UDFB or sub-program.



Press + to add a parameter and – to delete a parameter.



Change the order of the parameters between Categories and Alphabetical.



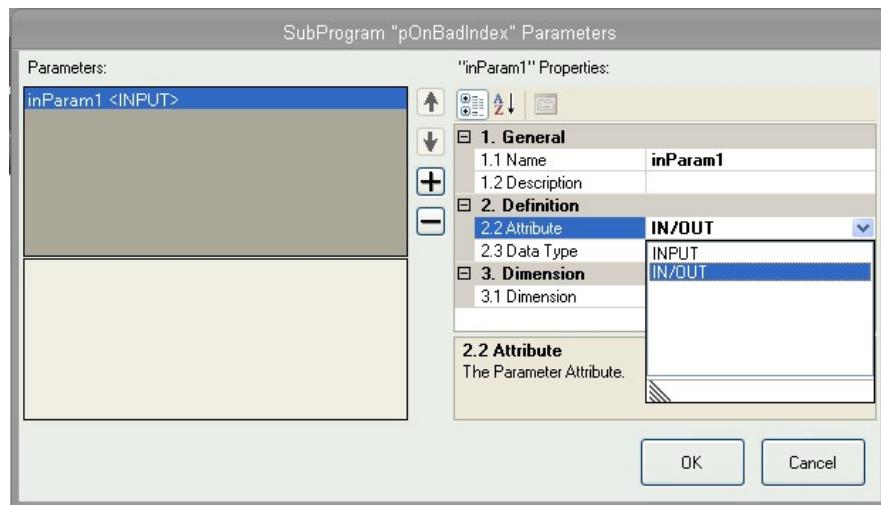
Input/output parameters can also be entered directly in the variable editor.

4. Press to add an input parameter, and under **General** in the right column, change its name to a more appropriate name.

If necessary, enter a description of the parameter.

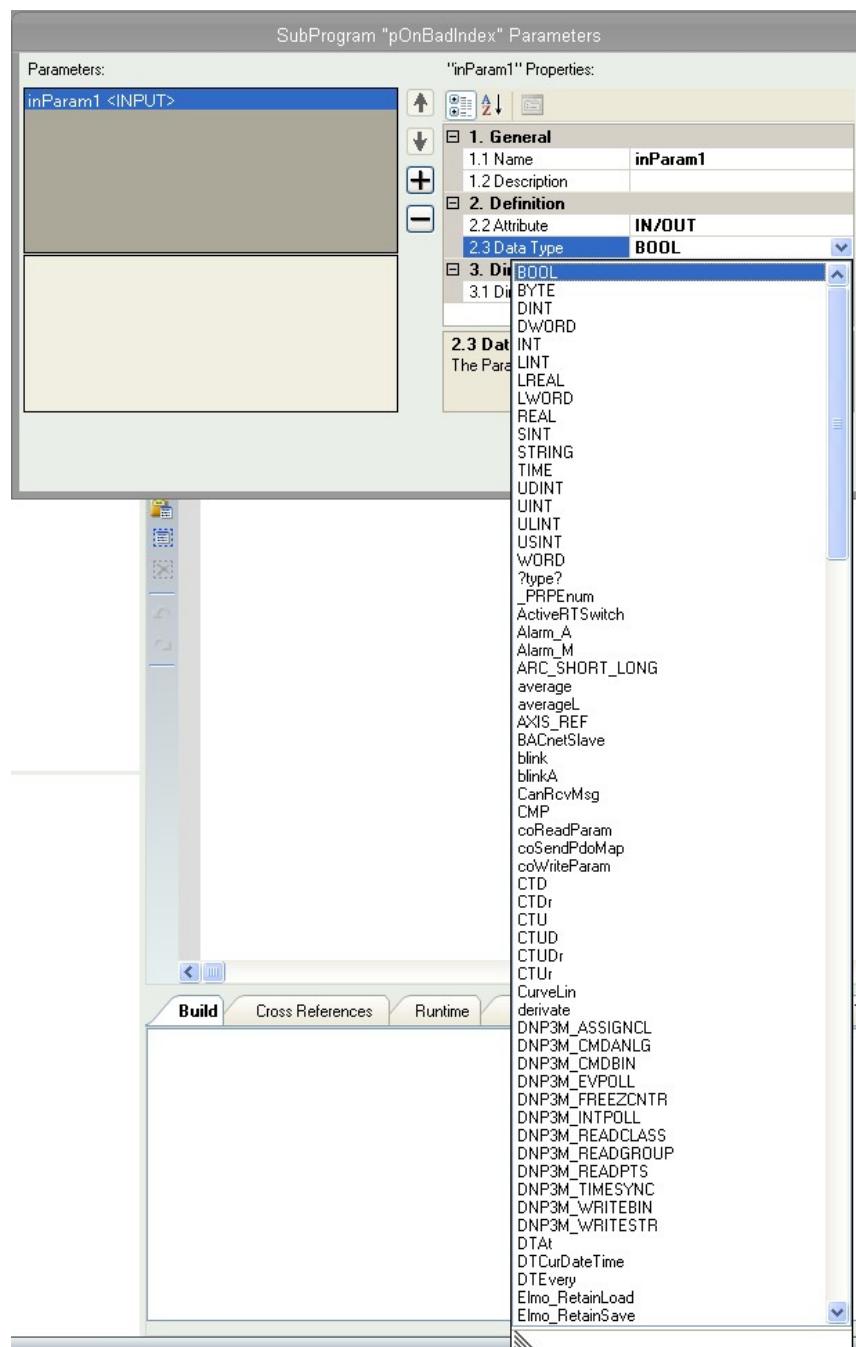
5. Under **Definition**, select from the available pull-down options:

Attribute





Data Type



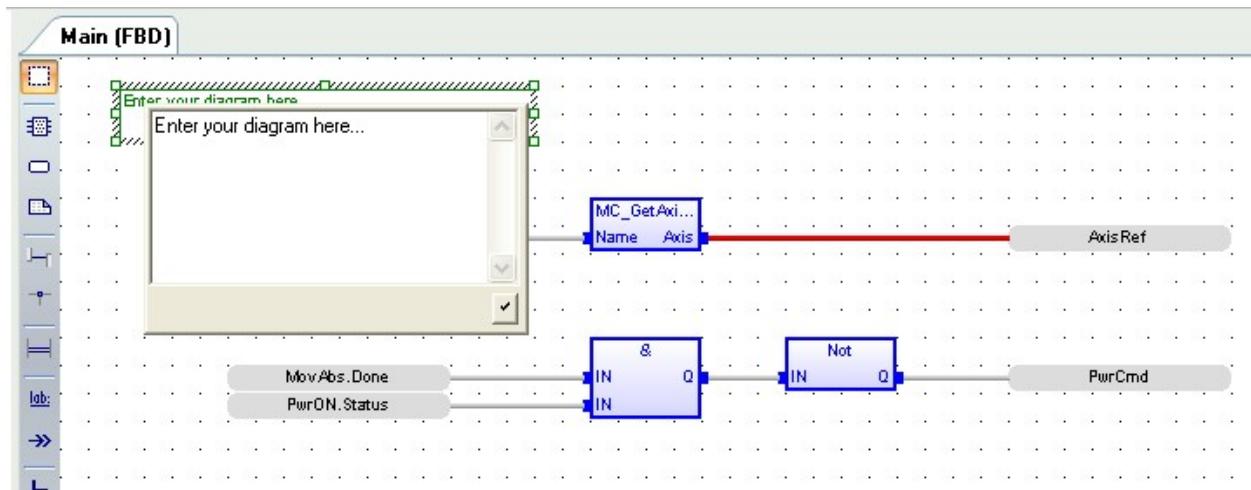
- Under **Dimension**, select from the available pull-down options:

Dimension Enter a relevant value

- Click **OK** to save the parameter(s) settings.



1.5.3.3. Description



In addition to the single line description entered in the main page for each POU, you can here enter a complete multiline description text where you can:

- Describe the behaviour and programming.
- Add some modification tracking notes.
- Add some guidelines if the POU is aimed to be used in other applications.



1.6. Introduction to Creating Programs

This chapter introduces and describes the five IEC61131-3 standard programming languages. For each program or application User Defined Function Block (UDFB) created, a specific language is selected. This chapter explains the differences between the various languages and helps in the selection of the appropriate language(s) for a specific project:

SFC	Sequential Function Chart
FBD	Function Block Diagram
LD	Ladder Diagram
ST	Structured Text , and the use of ST instructions in graphic languages
IL	Instruction List

1.6.1. Sequential Function Chart (SFC)

The SFC language is a state diagram. Graphical steps are used to represent stable states, and transitions describe the conditions and events that lead to a change of state. Using SFC highly simplifies the programming of sequential operations as it saves using variables and testing to maintain the program context.



Attention

SFC should not be used as a decision diagram as it results in poor performance and complicate charts. Therefore the SFC chart should never contain steps as points of decision with transitions as conditions. ST is the recommended language when programming a decision algorithm that is not relevant to the Program State.

The basic components of an SFC chart or program are:

Chart

Steps and initial steps

Transitions and divergences

Parallel branches

Macro-steps

Jump to a step

Programming

Actions within a step.

Timeout on a step

Programming a transition condition

How SFC is executed

UDFBs programmed in SFC



The workbench fully supports SFC programming with several hierarchical chart levels, i.e. a chart can control several other charts. This affords an easy and powerful method to manage complex sequences and save performance at run time. Refer to the sections 1.6.6 and 1.6.7 for further details.

1.6.1.1. SFC Steps

A step represents a stable state. It is drawn as a square box in the SFC chart. Each step of a program is identified by a unique number. At run time, a step can be either active or inactive according to the state of the program.



To change the number of a step, transition or jump, select it and hit **Ctrl+<Enter>** keys.

All actions linked to the steps are executed according to the activity of the step.

Inactive step



Active step



In SFC program conditions and actions, the step activity can be tested by specifying its name ("GS" plus the step number) followed by ".X".



Example

GS100.X Is *TRUE* if step 100 is active

(expression has the BOOL data type)

The activity time of a step can also be tested by specifying the step name followed by ".T". It is the time elapsed since the activation of the step. When the step is de-activated, this time remains unchanged. It will be reset to 0 on the next step activation.



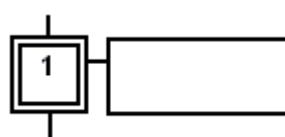
Example

GS100.T Is the time elapsed since step 100 was activated

(expression has the TIME data type)

1.6.1.1.a Initial steps

Initial steps represent the initial situation of the chart when the program is started. There must be at least one initial step in each SFC chart. An initial step is marked with a double line:





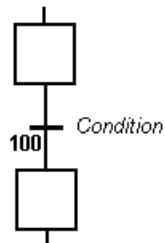
1.6.1.2. SFC Transitions

Transitions represent a condition that changes the program activity from one step to another.



To change the number of a step, transition or jump, select it and press <Ctrl>+<Enter> keys.

The transition is marked by a small horizontal line that crosses a link drawn between the two steps:



Each transition is identified by a unique number in the SFC program.

Each transition must be completed with a Boolean condition that indicates if the transition can be proceeded. The condition is a BOOL expression.

In order to simplify the chart and reduce the number of drawn links, you can specify the activity flag of a step (GSnnn.X) in the transition condition.

Transitions define the dynamic behaviour of the SFC chart, according to the following rules:

- A transition is proceeded if:

its condition is *TRUE*.

and if all steps linked to the top of the transition (before) are active.

- When a transition is proceeded:

all steps linked to the top of the transition (pre-transition) are de-activated.

all steps linked to the bottom of the transition (post-transition) are activated.

1.6.1.2.a Divergences

Several transitions can be linked to a step creating a Divergence. The divergence is represented by a horizontal line. Transitions after the divergence represent several possible changes in the state of the program.

All conditions are considered as exclusive, according to a *left-to-right* priority order. Therefore a transition is considered as *FALSE* if at least one of the transitions connected to the same divergence on its left side is *TRUE*.



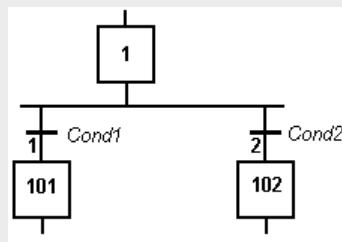
Example

Transition 1 is proceeded, if:

step 1 is active and Cond1 is *TRUE*

Transition 2 is proceeded, if:

step 1 is active and Cond2 is *TRUE*
and Cond1 is *FALSE*

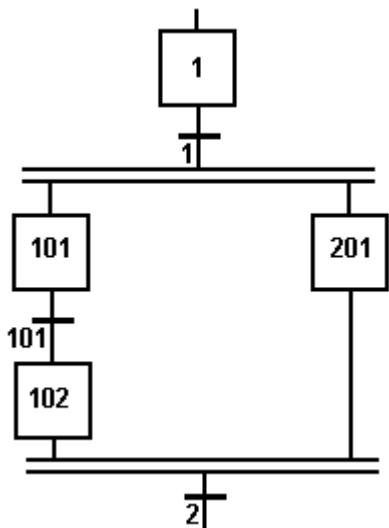


**Attention:**

Some run-time systems may not support exclusivity of the transitions within a divergence. Please refer to OEM instructions for further information about SFC support.

1.6.1.3. SFC parallel branches

Parallel branches are used in SFC charts to represent parallel operations. Parallel branches occur when more than several steps are connected after the same transition. Parallel branches are drawn as double horizontal lines:



When the transition before the divergence (1 in Figure 1) is proceeded, all steps beginning the parallel branches (101 and 201) are activated.

Processing of parallel branches may take different timing according to each branch execution.

The transition after the convergence (2 in Figure 1) is proceeded when all the steps connected before the convergence line (last step of each branch) are active. The transition indicates a synchronization of all parallel branches.

If necessary, a branch may be completed with an *Empty* step (with no action). It represents the state where the branch "waits" for the other ones to be completed.

Figure 1.1: SFC Parallel branch example

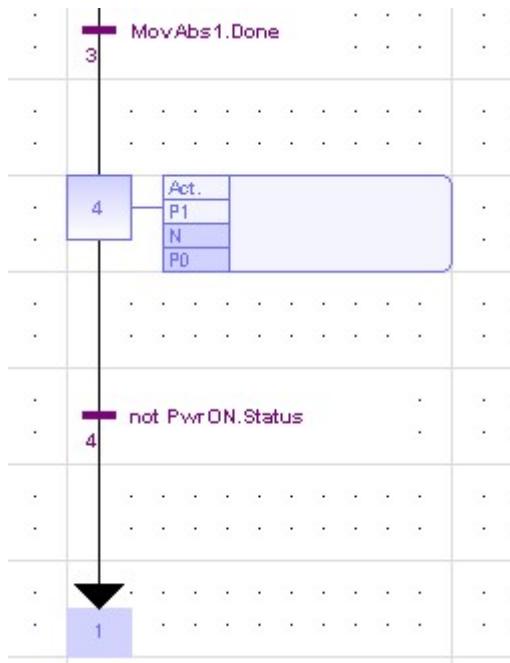
Make sure to follow the following guidelines when drawing parallel lines, to avoid dead locks in the execution of the program:

- All branches must be connected to the divergence and the convergence.
- An element of a branch must not be connected to an element outside the divergence.

1.6.2. Jump to a SFC step



Jump symbols can be used in SFC charts to represent a link from a transition to a step without actually drawing it. The jump is represented by an arrow identified with the number of the target step.

**Info:**

To change the number of a step, transition or jump, select it and press <Ctrl>+<Enter> keys.

Do not insert a jump to a transition which may lead to a non explicit convergence of parallel branches (several steps leading to the same transition). This will result in mistakes caused by an incorrect understanding of the chart.



1.6.3. Actions in a SFC step

Each step has a list of action blocks that are instructions to be executed according to the activity of the step. Actions can be simple Boolean or SFC actions consisting in the assignment of a Boolean variable, control of a child SFC program using the step activity, or action blocks entered using another language (FBD, LD, ST or IL).

1.6.3.1. Runtime Check

Use a syntax within an SFC step to perform runtime safety checks as shown in the example below:

Syntax	Description
<code>StepTimeout (...);</code>	Check for a timeout on the step activity duration

1.6.3.2. Simple Boolean Actions

Below are the possible syntaxes you can use within an SFC step to perform a simple Boolean action:

Syntax	Description
<code>BoolVar (N);</code>	Forces the variable <i>BoolVar</i> to <i>TRUE</i> when the step is activated, and to <i>FALSE</i> when the step is de-activated.
<code>BoolVar (S);</code>	Sets the variable <i>BoolVar</i> to <i>TRUE</i> when step is activated
<code>BoolVar (R);</code>	Sets the variable <i>BoolVar</i> to <i>FALSE</i> when step is activated
<code>/ BoolVar;</code>	Forces the variable <i>BoolVar</i> to <i>FALSE</i> when the step is activated, and to <i>TRUE</i> when the step is de-activated.

1.6.3.3. Alarms

The following syntax enables you to manage timeout alarm variables:

Syntax	Description
<code>BoolVar (A, duration);</code>	Specifies a timeout variable to be associated to the step. <ul style="list-style-type: none">• <i>BoolVar</i> must be a simple Boolean variable• <i>duration</i> is the timeout, expressed either as a constant or as a single TIME variable (complex expressions cannot be used for this parameter)

When the timeout is elapsed, the alarm variable is turned to *TRUE*, and the transition(s) following the step cannot proceed until the alarm variable is reset.

1.6.3.4. Simple SFC actions

Below are the possible syntaxes you can use within an SFC step to control a child SFC program:

Syntax	Description
<code>Child (N);</code>	Starts the child program when the step is activated and stops (kills) it when



the step is de-activated.

Child (S); Starts the child program when the step is activated

Child (R); Stops (kills) the child program when the step is activated

1.6.3.5. Programmed action blocks

Programs in other languages (FBD, LD, ST or IL) can be entered to describe an SFC step action. There are three main types of programmed action blocks, that correspond to the following identifiers:

Identifier	Description
P1	Executed only once when the step becomes active.
N	Executed on each cycle while the step is active.
P0	Executed only once when the step becomes inactive.

The workbench provides you with templates to enter P1, N and P0 action blocks in either ST, LD or FBD language. Alternatively, you can directly insert action blocks programmed in ST language in the list of simple actions, using the following syntax:

ACTION (qualifier) :

statements...

END_ACTION;

Where the *qualifier* is *P1*, *N* or *P0*.

1.6.3.6. Check timeout on a SFC step

The system can check the timeout on any SFC step activity duration by entering the following instruction in the main "Action" list of the step:

_StepTimeout (timeOut , errString);

Where:

timeOut is a time constant or a time variable specifying the timeout duration.

errString is a string constant or a string variable specifying the error message to be output.

At runtime, each time the activation time of the step becomes greater than the specified timeout, the error string is sent to the Workbench and displayed in the Log window.



Attention

Sending log message strings to the log window requires the runtime to be connected through ETHERNET, and that your EAS runtime system supports plain text trace messages.



You can also put this statement within a `#ifdef _DEBUG` test so that timeout checking is enabled only in debug mode.

Alternatively, if you need to manage specific timeouts, you can enter the following ST program in the "N" action block of the step:

```
if GSn.T > timeout then /* 'n' is the number of the step */  
    ...statements...  
end_if;
```

1.6.4. SFC Transition Conditions

Every SFC transition must have a Boolean condition that indicates if the transition can proceed. The condition is a Boolean expression that can be programmed either in ST or LD language.

In ST language, enter a Boolean expression. It can be a complex expression including function calls and parenthesis.



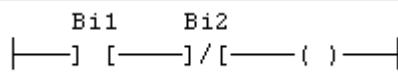
Example

`bForce AND (bAlarm OR min (iLevel, 1) <> 1)`

In LD language, the condition is represented by a single rung. The coil at the end of the rung represents the transition and should have no symbol attached.



Example



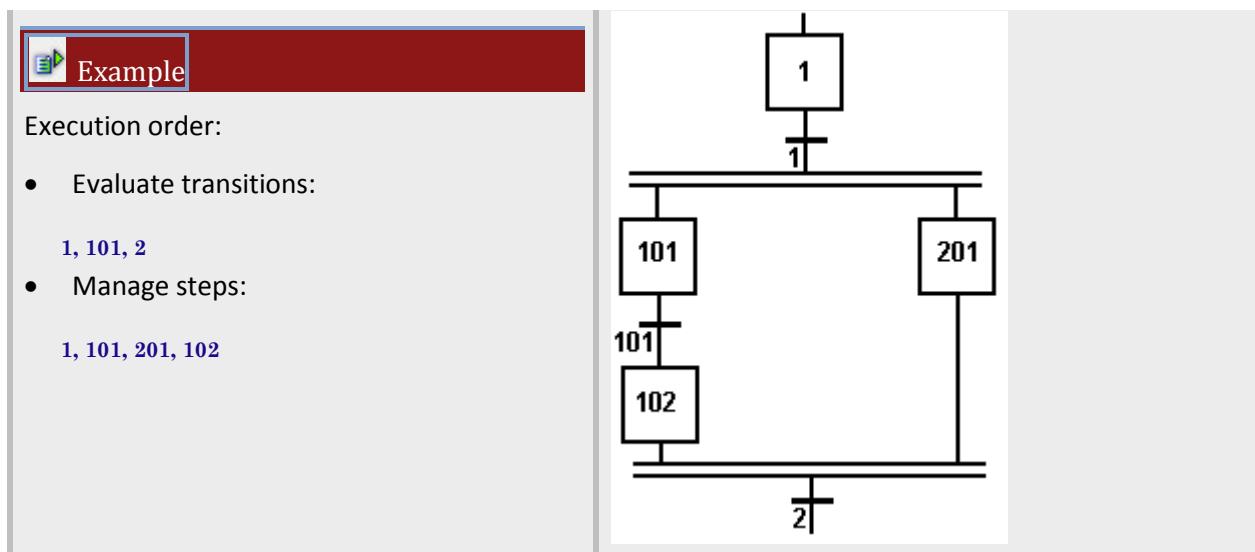


1.6.5. SFC execution at run time

SFC programs are executed sequentially within a target cycle, according to the order defined when entering programs in the hierarchy tree. A parent SFC program is executed before its children. This implies that when a parent starts or stops a child, the corresponding actions in the child program are performed during the same cycle.

This section describes the execution model of a standard target. SFC execution rules may differ for other target systems. Please refer to OEM instructions for further details about SFC execution at run time.

Within a chart, all valid transitions are evaluated first, and then actions of active steps are performed. The chart is evaluated from the left to the right and from the top to the bottom.



In case of a divergence, all conditions are considered as *exclusive*, according to a *left to right* priority order. It means that a transition is considered as *FALSE* if at least one of the transitions connected to the same divergence on its left side is *TRUE*.

The initial steps define the initial status of the program when it starts. All top level (main) programs are started when the application starts. Child programs explicitly start from action blocks within the parent programs.

The evaluation of transitions leads to changes of the active steps, according to the following rules:

- A transition proceeds, if:

Its condition is *TRUE*.

And if all steps linked to the top of the transition (before) are active.

- When a transition is proceeded:

All steps linked to the top of the transition (before) are de-activated.

All steps linked to the bottom of the transition (after) are activated.



Attention

Execution of SFC within the target is sampled according to the target cycles. When a transition proceeds within a cycle, the following steps are activated, and the evaluation of the chart continues on the next cycle. If several consecutive transitions are *TRUE* within a branch, only one of them proceeds within one target cycle.

Some run-time systems may not support exclusivity of the transitions within a divergence. Please refer to OEM instructions for further information about SFC support.

1.6.6. Hierarchy of SFC programs

Each SFC program may have one or more *child programs*. Child programs are written in SFC and are started (launched) or stopped (killed) in the actions of the parent program. A child program may also have children. The number of hierarchy levels should not exceed 19.

When a child program is stopped, its children are also implicitly stopped.

When a child program is started, it must explicitly start its children in its actions.

A child program is controlled (started or stopped) from the action blocks of its parent program.

Designing a child program is a simple way to program an action block in SFC language.

Using child programs is very useful for designing a complex process and separate operations due to different aspects of the process. For instance, it is common to manage the execution modes in a parent program and to handle details of the process operations in child programs.



1.6.7. Controlling a SFC child program

Controlling a child program may be simply achieved by specifying the name of the child program as an action block in a step of its parent program. Below are possible qualifiers that can be applied to an action block for handling a child program:

Qualifier	Description
Child (N);	Starts the child program when the step is activated and stops (kills) it when the step is de-activated.
Child (S);	Starts the child program when the step is activated. (Initial steps of the child program are activated)
Child (R);	Stops (kills) the child program when the step is activated. (All active steps of the child program are deactivated)

Alternatively, you can use the following statements in an action block programmed in ST language. In the following table, *prog* represents the name of the child program:

Statement	Description
GSTART (prog);	Starts the child program when the step is activated. (Initial steps of the child program are activated)
GKILL (prog);	Stops (kills) the child program when the step is activated. (All active steps of the child program are deactivated)
GFREEZE (prog);	Suspends the execution of a child program.
GRST (prog);	Restarts a program suspended by a GFREEZE command.

You can also use the GSTATUS function in expressions. This function returns the current state of a child SFC program:

Statement	Description
GSTATUS (prog);	Returns the current state of a child SFC program: 0: program is inactive 1: program is active 2: program is suspended



When a child program is started by its parent program, it retains its inactive status until it is executed (further in the cycle). If you start a child program in an SFC chart, GSTATUS will return 1 (active) on the next cycle.



1.6.8. User Defined Function Blocks programmed in SFC

The Workbench enables you to create User Defined Function Blocks (UDFBs) programmed with SFC language. This section details specific features related to such function blocks.

Declaration From the Workspace contextual menu, run the *Insert New Program* command. Then specify a valid name for the function block. Select "SFC" language and "UDFB" execution style.

Parameters When a UDFB programmed in SFC is created, the Workbench automatically declares 3 special inputs to the block:

RUN: The SFC state machine is not activated when this input is *FALSE*.

RESET: The SFC chart is reset to its initial situation when this input is *TRUE*.

KILL: Any active step of the SFC chart is deactivated when this input is *TRUE*.

You can freely add other input and output variables to the UDFB. You can also remove any of the automatically created input if not needed. If the *RUN* input is removed, then it is considered as always *TRUE*. If *RESET* or *KILL* inputs are removed, then they are considered as always *FALSE*.

Below is the truth table showing priorities among special input:

RUN	RESET	KILL	Description
<i>FALSE</i>	<i>FALSE</i>	<i>FALSE</i>	do nothing
<i>FALSE</i>	<i>FALSE</i>	<i>TRUE</i>	kill the SFC chart
<i>FALSE</i>	<i>TRUE</i>	<i>TRUE</i>	reset the SFC chart
<i>FALSE</i>	<i>TRUE</i>	<i>FALSE</i>	kill the SFC chart
<i>TRUE</i>	<i>FALSE</i>	<i>TRUE</i>	activate the SFC chart
<i>TRUE</i>	<i>FALSE</i>	<i>FALSE</i>	kill the SFC chart
<i>TRUE</i>	<i>TRUE</i>	<i>TRUE</i>	reset the SFC chart
<i>TRUE</i>	<i>TRUE</i>	<i>FALSE</i>	kill the SFC chart
Steps	All steps inserted in the SFC chart of the UDFB are automatically declared as local instances of special reserved function blocks with the local variables of the UDFBs. The following FB types are used:		
	<i>isfcSTEP</i> : a normal step		
	<i>isfcINITSTEP</i> : an initial step		
	The editor takes care of updating the list of declared step instances. You should never remove, rename or change them in the variable editor. All steps are named with <i>GS</i> followed by their number.		

**Execution**

The SFC chart is operated only when the UDFB is called by its parent program.

If the *RESET* input is *TRUE*, the SFC chart is reset to its initial situation. If the *KILL* input is *TRUE*, any active step of the SFC chart is deactivated.

When the *RUN* input is *TRUE* and *KILL/RESET* are *FALSE*, the SFC chart is operated in the same way as for other SFC programs:

1. Check valid transitions and evaluate related conditions.
2. Cross *TRUE* valid transitions.
3. Execute relevant actions of the active steps.

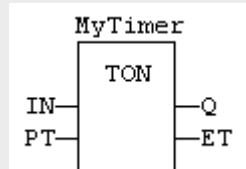


In a UDFB programmed in SFC, you cannot use SFC actions to pilot a "child SFC program". This feature is reserved for SFC programs only. Instead, a UDFB programmed in SFC can pilot from its actions another UDFB programmed in SFC.

1.6.9. Function Block Diagram (FBD)

A Function Block Diagram is a data flow between constant expressions or variables and operations represented by rectangular blocks. Operations can be basic operations, function calls, or function block calls.

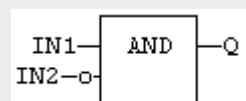
The name of the operation, function, or type of function block, is written within the block rectangle. In the case of a function block call, the name of the called instance must be written upon the block rectangle, as shown below:

**Example**

The data flow may represent values of any data type. All connections must be from input and outputs points having the same data type. In case of a Boolean connection, you can use a connection link terminated by a small circle, indicating a Boolean negation of the data flow.

**Example**

Use of a negated link: Q is IN1 AND NOT IN2!



The data flow must be understood from the left to the right and from the top to the bottom. It is possible to use labels and jumps to change the default data flow execution.

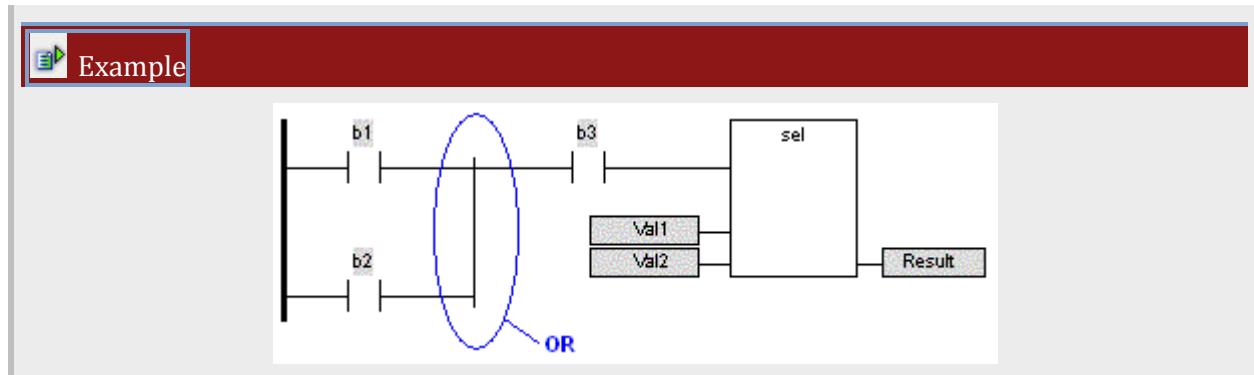


1.6.9.1. LD symbols

LD symbols may also be entered in FBD diagrams and linked to FBD objects. Refer to the following sections for further Information about components of the LD language:

- Contacts
- Coils
- Power Rails

Special vertical lines are available in FBD language for representing the merging of LD parallel lines. Such vertical lines represent a OR operation between the connected inputs. Below is an example of an OR vertical line used in a FBD diagram.





1.6.10. Ladder Diagram (LD)

A Ladder Diagram is a list of rungs. Each rung represents a Boolean data flow from a power rail on the left to a power rail on the right. The left power rail represents the *TRUE* state. The data flow must be understood from the left to the right. Each symbol connected to the rung either changes the rung state or performs an operation. Below are possible graphic items to be entered in LD diagrams:

- Power Rails
- Contacts and Coils
- Operations, Functions and Function blocks, represented by rectangular blocks
- Labels and Jumps
- Use of ST instructions in graphic languages

1.6.10.1. Use of EN Input and ENO Output for Blocks

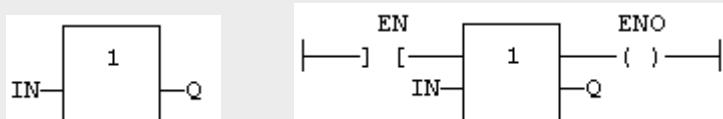
The rung state in a LD diagram is always Boolean. Blocks are connected to the rung with their first input and output. This implies that special *EN*, *ENO*, input and output are added to the block if its initial input or output is not Boolean.

The *EN* input is a condition. It means that the operation represented by the block is not performed if the rung state (*EN*) is *FALSE*. The *ENO* output always represents the same status as the *EN* input; the rung state is not modified by a block having an *ENO* output.

The following examples are of the XOR block, with Boolean inputs and outputs, and requiring no *EN* or *ENO* pin:



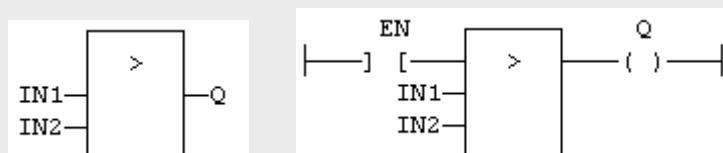
First input is the rung. The rung is the output.



Below is the Example of the > (greater than) block, with non-Boolean inputs and a Boolean output. This block has an *EN* input in LD language:



The comparison is executed only if *EN* is *TRUE*.



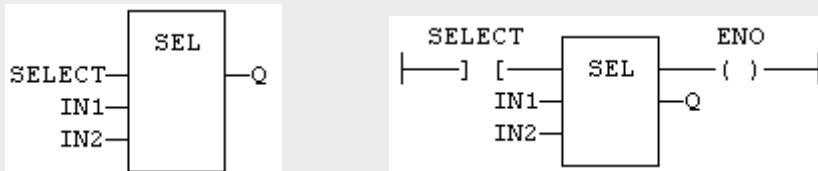
Below is the Example of the SEL function, with a first Boolean input, but an integer output. This block has an *ENO* output in LD language:



Info

The input rung is the selector.

ENO has the same value as *SELECT*.

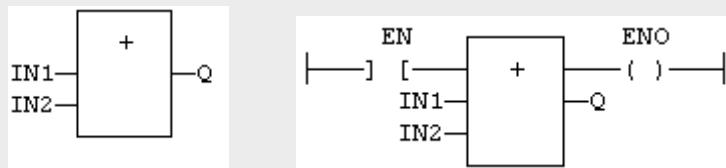


Finally, below is the Example of an addition, with only numerical arguments. This block has both *EN* and *ENO* pins in LD language:

Info

The addition is executed only if *EN* is *TRUE*.

ENO is equal to *EN*.



1.6.10.2. Contacts

Contacts are basic graphic elements of the LD language. A contact is associated with a Boolean variable written above the graphic symbol. A contact sets the state of the rung on its right side, according to the value of the associated variable and the rung state on its left side.

Below are the possible contact symbols and their rung state change:

Symbol	Description
BoolVariable —] [—	Normal: the rung state on the right is the Boolean AND between the rung state on the left and the associated variable.
BoolVariable —]/[—	Negated: the rung state on the right is the Boolean AND between the rung state on the left and the negation of the associated variable.
BoolVariable —] P [—	Positive pulse: the rung state on the right is TRUE only when the rung state on the left is TRUE and the associated variable changes from FALSE to TRUE (rising edge).
BoolVariable —] N [—	Negative pulse: the rung state on the right is TRUE only when the rung state on the left is TRUE and the associated variable changes from TRUE to FALSE (falling edge).

Info

When a contact or a coil is selected, press the SPACE bar to change its type (normal,



negated, pulse.).

Two serial normal contacts represent an AND operation.

Two contacts in parallel represent an OR operation.

1.6.10.3. Coils

Coils are basic graphic elements of the LD language. A coil is associated with a Boolean variable described above its graphic symbol. A coil performs a change of the associated variable according to the rung state on its left side.

Below are the possible coil symbols and their rung state change:

Symbol	Description
BooVariable 	<i>Normal</i> : the associated variable is forced to the value of the rung state on the left of the coil.
BooVariable 	<i>Negated</i> : the associated variable is forced to the negation of the rung state on the left of the coil.
BooVariable 	<i>Set</i> : the associated variable is forced to <i>TRUE</i> if the rung state on the left is <i>TRUE</i> . (no action if the rung state is <i>FALSE</i>)
BooVariable 	<i>Reset</i> : the associated variable is forced to <i>FALSE</i> if the rung state on the left is <i>TRUE</i> . (no action if the rung state is <i>FALSE</i>)



When a contact or a coil is selected, press the SPACE bar to change its type (normal, negated, pulse.).



Even though coils are commonly connected to a power rail on the right, the rung may be continued after a coil. The rung state is never changed by a coil symbol.

1.6.10.4. Power Rails

Vertical power rails are used in LD language to represent the limits of a rung. The power rail on the left represents the *TRUE* value and initiates the rung state. The power rail on the right receives connections from the coils and has no influence on the execution of the program.

Power rails can also be used in FBD language. Only Boolean objects can be connected to left and right power rails.



1.6.11. Structured Text (ST)

ST is a structured literal programming language. A ST program is a list of statements. Each statement describes an action and must end with a semicolon (";").

The presentation of the text has no meaning for a ST program. You can insert blank characters and line breaks where you want in the program text.

Comments

Comment texts can be entered anywhere in a ST program. Comment texts have no meaning for the execution of the program. A comment text must begin with "(*" and end with "*)". Comments can be entered on several lines (i.e. a comment text may include line breaks). Comment texts cannot be nested.

```
(* My comment *)
a := d + e;

(* A comment can also
be on several lines *)
b := d * e;

c := d - e; (* My comment *)
```

Expressions

Each statement describes an action and may include evaluation of complex expressions. An expression is evaluated:

- From the left to the right
- According to the default priority order of operators

Operators

- (...) NOT (...)
** (power)
* /
+ -
< > <= >= <> = (comparisons)
AND (you can use "&")
OR
XOR



- The default priority can be changed using parenthesis

Arguments of an expression can be:

- Declared variables
- Constant expressions
- Function calls

Statements

The following basic statements that can be entered in a ST program:

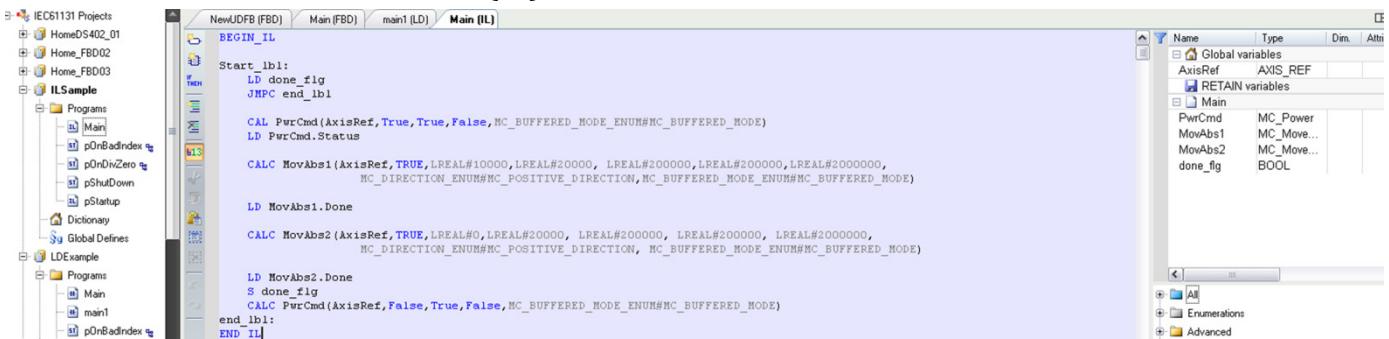
- Assignment
- Function block calling
- The following available conditional statements in ST language:
IF / THEN / ELSE (simple binary switch)
CASE (enumerated switch)

Below are the available statements for describing loops in ST language:

- WHILE (with test on loop entry)
- REPEAT (with test on loop exit)
- FOR (enumeration)



1.6.12. Instruction List (IL)



A program written in IL language is a list of *instructions*. Each instruction is written on one line of text. An instruction may have one or more **operands**. Operands are variables or constant expressions. Each instruction may begin with a label, followed by the ":" character. Labels are used as destination for jump instructions.

The Workbench allows mixing of ST and IL languages in textual program. ST is the default language. When IL instructions are entered, the program must be entered between BEGIN_IL and END_IL keywords as in the following example:

Example

```
BEGIN_IL
LD var1
ST var2
END_IL
```

Comments Comment texts can be entered at the end of a line containing an instruction. Comment texts have no meaning for the execution of the program. A comment text must begin with "(" and end with ")". Comments may also be entered on empty lines (with no instruction), and on several lines (i.e. a comment text may include line breaks). Comment texts cannot be nested.

Data flow An IL complete statement is made of instructions for:

- first: evaluating an expression (called *current result*)
- then: use the current result for performing actions

Evaluation of expressions The order of instructions in the program is the one used for evaluating expressions, unless parentheses are inserted. Below are the available instructions for evaluation of expressions:

Instruction	Operand	Meaning
LD / LDN	any type	Loads the operand in the current result
AND	Boolean	AND between the operand and the current result.



OR / ORN	Boolean	OR between the operand and the current result.
XOR / XORN	Boolean	XOR between the operand and the current result.
ADD	numerical	Adds the operand and the current result.
SUB	numerical	Subtract the operand from the current result.
MUL	numerical	Multiply the operand and the current result.
DIV	numerical	Divide the current result by the operand.
GT	numerical	Compares the result with the operand.
GE	numerical	Compares the result with the operand.
LT	numerical	Compares the result with the operand.
LE	numerical	Compares the result with the operand.
EQ	numerical	Compares the result with the operand.
NE	numerical	Compares the result with the operand.
Function call	func. arguments	Calls a function.
Parenthesis		Changes the execution folder.



Instructions suffixed by N uses the Boolean negation of the operand.

Actions The following instructions perform actions according to the value of current result. Some of these instructions do not need a current result to be evaluated:

Instruction	Operand	Meaning
ST / STN	any type	Stores the current result in the operand.
JMP	label	Jump to a label - no current result needed.
JMPC	label	Jump to a label if the current result is <i>TRUE</i> .
JMPNC / JMPCN	label	Jump to a label if the current result is <i>FALSE</i> .
S	Boolean	Sets the operand to <i>TRUE</i> if the current result is <i>TRUE</i> .
R	Boolean	Sets the operand to <i>FALSE</i> if the current result is <i>TRUE</i> .
CAL	f. block	Calls a function block (no current result needed).
CALC	f. block	Calls a function block if the current result is <i>TRUE</i> .
CALNC / CALCN	f. block	Calls a function block if the current result is <i>FALSE</i> .



Instructions suffixed by N uses the Boolean negation of the operand.

1.6.13. Use of ST expressions in graphic language

The workbench allows any complex ST expression to be associated with a graphic element in either LD or FBD language. This feature makes it possible to simplify LD and FBD diagrams when some trivial calculation has to be entered. It also enables you to use graphic features to represent a main algorithm as text is used for details of implementation.

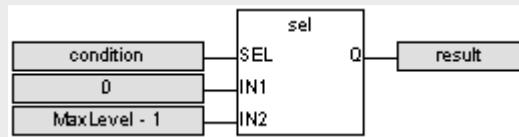
Expression must be written in ST language. An expression is any logical expression structured between parenthesis in a ST program. Obviously the ST expression must fit the data type required by the diagram (e.g. an expression put on a contact must be Boolean).

1.6.13.1. FBD Language

A complex ST expression can be entered in any *variable box* of a FBD diagram, if the box is not connected to its input.



Example

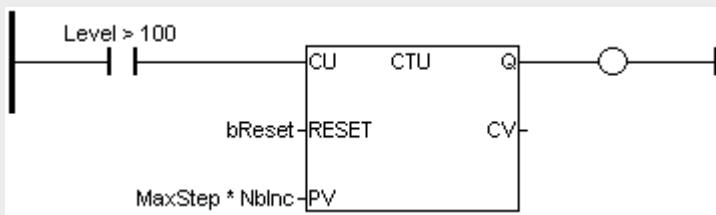


1.6.13.2. LD Language

A complex ST expression can be entered on any kind of contact, and on any input of a function or function block.



Example





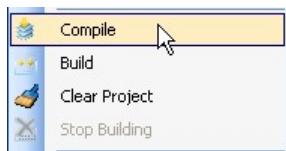
1.7. Building or Compiling the project

The project must be built (or compiled) before it is simulated or downloaded to the target. When either Compile or Build is selected the compiler reports messages display in the Build tab below the main display window. If compiling errors occur, just double click on an error message to highlight the corresponding function / function block error in the opened program at the appropriate location.

Compiling can be performed whenever the project is attached to the WorkSpace. However, Building can only be performed when the project is linked to the G-MAS, or other motion controller.

To Compile a project:

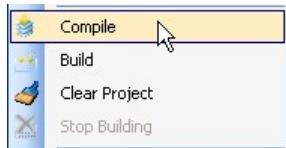
1. Right-click on the project and select **Compile** from the menu, or alternative click the icon .



The option Build is only available when the project is linked to the operational G-MAS target.

To Build a project:

1. Right-click on the project and select **Build** from the open menu, or alternative click the icon .





1.8. Variable editor

Dictionary		Graphic		Properties					
Name	Type	Dim.	Attrib.	Syb.	Init value	User Group	Tag	Description	
Global variables									
myAxis	AXIS_REF				<input type="checkbox"/>				
Inst_TON	TON				<input type="checkbox"/>				
Inst_MC_Stop	MC_Stop				<input type="checkbox"/>				
syopDonne	BOOL				<input type="checkbox"/>				
movAbsRepBusy	BOOL				<input type="checkbox"/>				
ChangeOPErrID	INT				<input type="checkbox"/>				
movAbsRepDone	BOOL				<input type="checkbox"/>				
movAbsRepErr	BOOL				<input type="checkbox"/>				
movAbsErrID	INT				<input type="checkbox"/>				
opModeArry	OP_MODE_DS402	[0..2]			<input type="checkbox"/>	OPM402_INTERPOLATED_P...			
z	DINT				<input type="checkbox"/>	0			
Inst_CTUr	CTUr				<input type="checkbox"/>				
reset	BOOL				<input type="checkbox"/>				
Inst_blink	blink				<input type="checkbox"/>				
powerVAlid	BOOL				<input type="checkbox"/>				
powerErr	BOOL				<input type="checkbox"/>				
powerErrID	INT				<input type="checkbox"/>				
RETAIN variables									
Main									
power	MC_Power				<input type="checkbox"/>				
start	BOOL				<input type="checkbox"/>	True			
changeOP	MC_ChangeOpMode				<input type="checkbox"/>				
moveAbsREP	MC_MoveAbsRep				<input type="checkbox"/>				

Variables are declared in the top/right area of the Workbench main window. The variable editor is a grid tool enabling all variables of the application to be declared. Variables in the editor are sorted by groups:

- Global variables.
- "Retain" non volatile global variables.
- I/O variables (each I/O device is a group).
- variables local to a program (including in and out parameters in case of a UDFB).

Refer to the description of the individual variables in chapter 3 Programming languages - Reference guide on page 157 for a more detailed overview.

Each group is marked with a gray header in the variable list. The "-" or "+" icon on the left of the group header can be used to expand or collapse the group:

Dictionary		Graphic		Properties					
Name	Type	Dim.	Attrib.	Syb.	Init value	User Group	Tag	Description	
Global variables									
RETAIN variables									
Main									



To sort, show or hide columns, and apply a filter for each column, double click on the header line. The filter is described as a text string that may contain '?' and '*' wildcharacters.

Use	Details
Using the grid to:	Create new variables Using the editing grid Sort variables of a group Define I/O devices Edit as text Bookmarks
Each variable is described with	A name A data type and a dimension An attribute An initial value A tag and a description text OEM defined properties A user group. The user group enables logical sorting of variables in the grid.  Info: Use the Syb column to specify if the variable symbol must be embedded, such as defined in the variable properties.

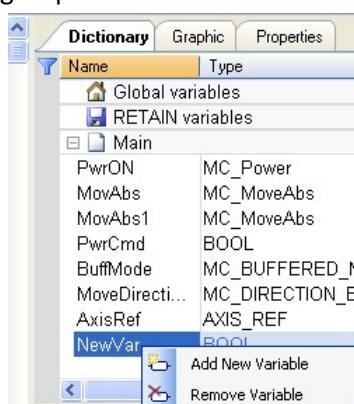


1.8.1. Creating new variables

To create a new variable:

1. Press the <Insert> key in the variable editor to create a new variable in the selected group.

Alternatively, right-click in the group and select **Add New Variable**.



The variable is added at the end of the group. Variables are created with a default name. You can rename a new variable or change its attribute using the variable editing grid. You cannot insert a new variable in an I/O group.

2. To remove the variable, right-click in the group and select **Remove Variable**.

Dictionary		Graphic	Properties		Dim.	Attrib.	Syb.	Init value	User Group	Tag	Description
Name	Type										
Global variables											
RETAIN variables											
Main											
PwrON	MC_Power										
MovAbs	MC_MoveAbs										
MovAbs1	MC_MoveAbs										
PwrCmd	BOOL										
BuffMode	MC_BUFFERED_MODE_ENUM										
MoveDirecti...	MC_DIRECTION_ENUM										
AxisRef	AXIS_REF										
NewVar	BOOL										

3. If the selected variables group corresponds to local UDFB variable, then pressing the <Insert> key provides with the selection between:
 - Adding an IN (input) parameter.
 - Adding an OUT (output) parameter.
 - Adding a private variable.

Dictionary		Graphic	Properties		Dim.	Attrib.	Syb.	Init value	User ...	Tag	Description
Name	Type										
Global variables											
RETAIN variables											
MyProgram											
input	BOOL		IN								
output	BOOL		INOUT								

IN and OUT parameters always appear at the beginning of a UDFB group.



1.8.2. Variable list

The variable editor enables you to enter directly each piece information in the cells of a grid. The name of the selected column is highlighted in yellow. The text of selected cell (or "..." if empty) is highlighted.

Dictionary		Graphic	Properties	Dim.	Attrib.	Syb.	Init value	User Group	Tag	Description
Name	Type									
Global variables										
RETAIN variables										
Main										
PwrON	MC_Power	[0..4]								
MovAbs	MC_MoveAbs									
MovAbs1	MC_MoveAbs									
PwrCmd	BOOL									
BuffMode	MC_BUFFERED_MODE_ENUM									
MoveDirecti...	MC_DIRECTION_ENUM									
AxisRef	AXIS_REF									

Dragging with the mouse, within the Variables grid moves the column separators in the main grid header for resizing columns.

Press the following keys for browsing groups of variables:

Shortcut	Description
----------	-------------

Ctrl + Page Up Move the selection to the head of the previous group.

Ctrl + Page Down Move the selection to the head of the following group.

Variables of a group can be sorted according to name, type, or dimension.

To sort a group's variables:

1. Move the cursor to the header of the group.

Dictionary		Graphic	Properties	Dim.	Attrib.	Syb.	Init value	User ...	Tag	Description
Name	Type									
Global variables										
RETAIN variables										
Main										
PwrON	MC_Power	[0..4]								
PwrCmd	BOOL									
MoveDirecti...	MC_DIRECTION_ENUM									
MovAbs1	MC_MoveAbs									
MovAbs	MC_MoveAbs									
BuffMode	MC_BUFFERED_MODE_ENUM									
AxisRef	AXIS_REF									

2. Click on the name of the desired column.

The workbench always keeps the original order of declared variables, in order to allow safe On Line change. Each time a new variable is inserted or a group expanded/collapsed, the original sorting is re-applied.



Dictionary		Graphic	Properties					
Name	Type	Dim.	Attrib.	Syb.	Init value	User ...	Tag	Description
Global variables								
RETAIN variables								
Main								
AxisRef	AXIS_REF							
BuffMode	MC_BUFF...							
MovAbs	MC_Move...							
MovAbs1	MC_Move...							
MoveDirecti...	MC_DIRE...							
PwrCmd	BOOL							
PwrON	MC_Power	[0..4]						

A variable must be identified by a unique name within its parent group. The variable name cannot be a reserved keyword of the programming languages and cannot have the same name as a standard or "C" function or function block. A variable should not have the same name as a program or a user defined function block.

The name of a variable should begin by a letter or an underscore ("_") mark, followed by letters, digits or underscore marks. It is not allowed to put two consecutive underscores within a variable name. Naming is case insensitive. Two names with different cases are considered as the same.

To rename a variable:

1. Move the cursor to the selected name cell.
2. Then press <Enter> or the first character of the new name. The name is entered in a textbox.

Dictionary		Graphic	Properties					
Name	Type	Dim.	Attrib.	Syb.	Init value	User ...	Tag	Description
Global variables								
RETAIN variables								
Main								
PwrON	MC_Power	[0..4]						
MovAbs	MC_Move...							
MovAbs1	MC_Move...							
PwrCmd	BOOL							
BuffMode	MC_BUFF...							
MoveDirecti...	MC_DIRE...							
AxisRef	AXIS_REF							
	AxisRef							

3. Press <Enter> to validate the name or <Escape> to cancel the change.

To change the type and dimension of the variable:

1. Move the cursor to the appropriate cell and press <Enter>. Each variable must have a valid data type. It can be either a basic data type or a type of function block or UDFB. Changing the data type of an I/O variable is restricted by the physical implementation of the I/O device.

Dictionary		Graphic	Properties					
Name	Type	Dim.	Attrib.	Syb.	Init value	User ...	Tag	Description
Global variables								
RETAIN variables								
NewPrg								
input_new	BOOL			IN				
output_new	BOOL			OUT				



2. If the selected data type is a STRING, specify a maximum length, that cannot exceed 255 characters.
3. Optionally the dimension(s) for an internal variable can be specified, in order to declare an array. Arrays have at most three dimensions. All indexes are 0 based. For instance, in case of single dimension array, the first element is always identified by ArrayName[0]. The total number of items in an array (merging all dimensions) cannot exceed 65535.
4. Enter integer dimensions separated by commas, using the variable editor. For instance: 3,10,5.



1.8.3. Variable attributes

Each variable has an attribute displayed in the corresponding column of the grid. Physical I/Os are marked as either *Input* or *Output*. Inputs are read-only variables. For each internal variable, Read Only is selectable. Otherwise, the attribute column of an internal variable is empty. Parameters of User Defined Function Blocks are marked as either *IN* or *OUT*.

Name	Type	Dim.	Attrib.	Syb.	Init value	User ...	Tag	Description
Global variables								
RETAIN variables								
NewPrg								
input_new	BOOL		IN	<input checked="" type="checkbox"/>				
output_new	BOOL		OUT	<input type="checkbox"/>				

To change the attribute of an internal variable:

1. Move the cursor to the selected attribute cell. Then press <Enter> to set or reset the Read Only attribute.

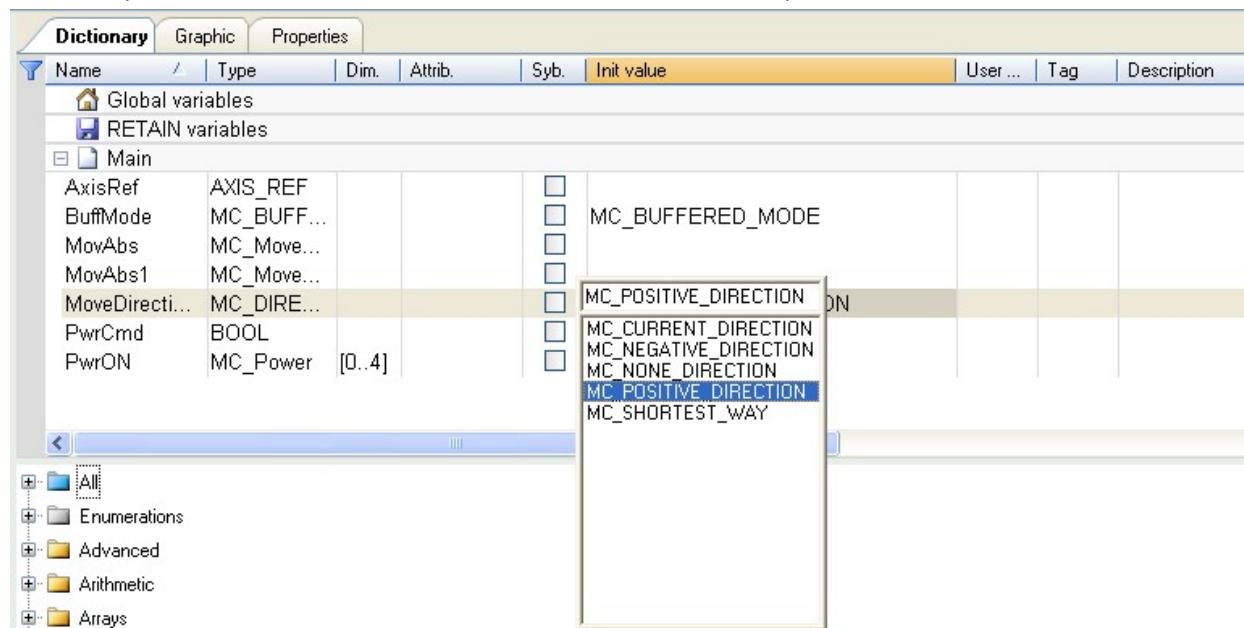


1.8.4. Initial value of a variable

A variable may have an initial value. The value must be a valid constant expression that complies with the variable data type. If the initial value is not a valid expression for the selected data type, it is displayed in red. There is no initial value for arrays and instances of function blocks.

To change the initial value of a variable:

1. Move the cursor to the selected init value cell. Then press <Enter> to select a new variable from the pull-down menu, or, if the variable has no Initial Value, press <Enter> and insert a new value.





1.8.5. Variable tag and Description

The workbench allows entering of two types of strings for each variable, to describe it:

- The Tag is a short comment, that can be displayed together with the variable name in graphic languages.
- The Description is a long comment text that describes the variable.

To change the tag or description the of a variable:

1. Move the cursor to the corresponding cell. Then press <Enter> to enter the new text.



1.9. Editing programs

The Programming environment provides dedicated language Editors for:

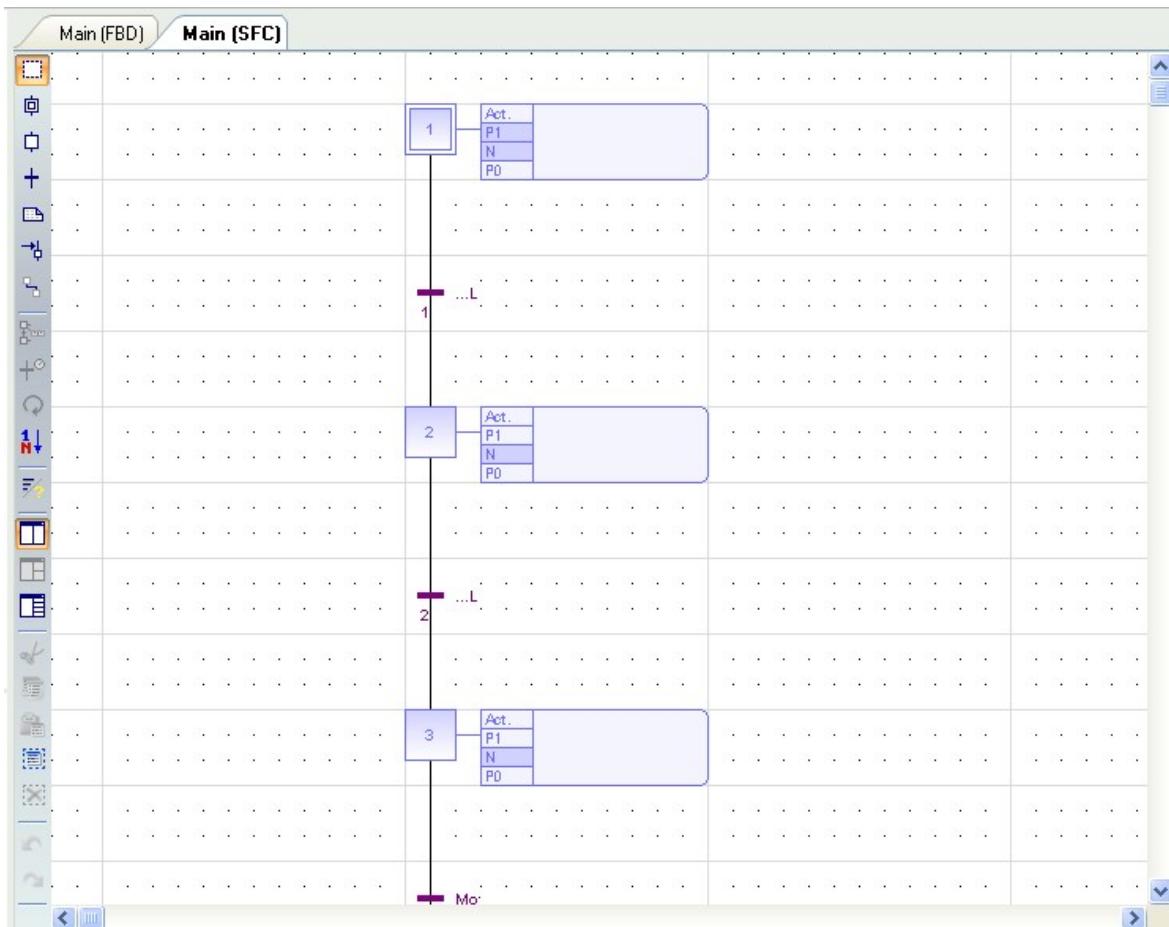
- Sequential Function Chart (SFC)
- Function Block Diagram (FBD)
- Ladder Diagram (LD)
- Structure Text (ST) and Instruction List (IL)

Each of these Editors provides the programming environment with drag and drop features:

- Drag a variable from the list to the program to insert it.
- Drag a definition to the program to insert its name.
- Drag a block in the program to insert it.
- Drag a function block to the variable list to declare an instance.
- Drag a variable from the program or from the variable list to the spy list.
- Double click on a line of the output window to highlight the corresponding code.



1.9.1. Sequential Function Chart (SFC) Editor



The Workbench consists of an editor to enter Sequential Function Chart (SFC) programs:

Free Form Editor The Free form editor supports advanced graphic features, freeing the user to arrange steps and transitions within a graphic layout.



The editor is selected when creating the program

SFC diagram components

Steps

Related sections

Using the SFC toolbar

Transitions

Drawing divergences

Divergences

Viewing the chart

Parallel branches

Moving or copying parts of the chart

Jump to a step

Entering macro-steps

Macro steps

Renumbering steps and transitions

Actions

Entering actions of a step

Renumbering steps and transitions

Conditions

Entering condition of a transition



Timeout check

Notes for steps and transitions

Bookmarks



Info:

To change the number of a step, transition or jump, select it and press the <Ctrl>+<Enter> keys. Press the <Space> bar when the pointer is on the main corner (on the left) of a divergence or convergence to set double/single horizontal line style.

1.9.1.1. SFC Free Form Editor

The SFC Free Form editor is a full graphical editor that allows management of the Sequential Function Chart. The editor supports advanced graphic features, enabling easy arrangement of steps and transitions within a graphic layout.

Simply drag items (step, transition, jump) from the editor toolbar to the graphic layout to insert objects. Objects can then be freely moved.

Draw links by dragging the pointer from an item to another in the direction of the flow. Placing the cursor over a step or transition, displays a green mark to indicate the source of the new dragged link.

To draw a divergence or convergence, simply link items together (for instance, from a step to several transitions). The editor automatically draws the horizontal lines. When a link is selected, its segments can be moved to change its routing.

1.9.1.2. Using the SFC toolbar

The vertical toolbar on the left side of the editor contains buttons for inserting items in the chart. Items are always inserted before the selected item. The chart is automatically rearranged when a new item is inserted.

Icon Description

- Selection mode. To edit, copy, paste, or remove, a function block, function, or item, specify the objects using the selection mode.
- Insert an initial step. This always the first and starting step in setting up any new chart.
- Insert a Step. Inserts an intermediate step at any point.
- Insert a transition. Inserts a transition
- Insert comment text: Use the mouse to insert comment text areas in the diagram. Comment texts can be entered anywhere. Click in the diagram and drag the text block to the desired position. The text area can then be selected and resized.
- Insert a jump to a step
- Add arc mode
- Edit a reference



Set the Timer on Transition



Replace the item style



Renumber steps and transitions



Switch between possible display overviews:

- Display no code or notes
- Display code of actions and conditions
- Display notes attached to steps and transitions



Show one qualifier in SFC level 2



Show two qualifiers in SFC level 2



Show all qualifiers in SFC level 2



Cut



Copy



Paste



Select All



Delete selection



Undo



Redo

Use the following keyboard commands when an item is selected:

<Enter> Edit the level 2 of a step or transition

<Ctrl>+<Enter> Change the number of a step, transition or jump

1.9.1.3. Moving and copying SFC charts

The SFC editor fully supports drag and drop to move or copy items.

To move items, select and drag them to the desired position.

To copy items, do the same, and just press the <CONTROL> key while dragging. It is also possible to drag pieces of charts from a program to another if both are open and visible on the screen.

To cancel any of the above operations while dragging items, press <ESCAPE>.

Alternatively, Use the classical **Copy / Cut / Paste** commands from the Edit menu. Paste is performed at the current position.



1.9.1.4. Renumbering steps and transitions

Each step or transition is identified by a number. A jump to a step is also identified by the number of the destination step. The SFC editor allocates a new number to each step or transition inserted in the chart.

To change the number of a step, transition or jump, select it and press the <Ctrl>+<Enter> keys.

It is not possible to change the number of a step or a transition if its level 2 is currently open for editing. The number is used for identifying the step or transition in the level 2 editing window.

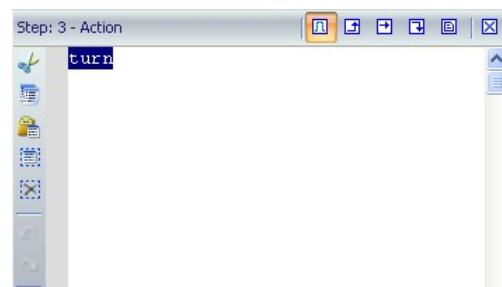
In the compiler reports, a step is identified by its number, prefixed by *GS*. A transition is identified by its number, prefixed by *GT*.

1.9.1.5. Entering step actions

Actions and notes attached to a step (level 2) are entered in a separate window. To open the level 2 editing window of a step or transition, double click on its symbol in the chart, or select it and press <Enter>. The level 2 editing window consists of four views to enter various types of level 2 information:

Simple actions entered as text. The menu bar at the left side is detailed below.

Notice that the first top bar icon is highlighted. This is the present level 2 mode.



Cut



Copy



Paste



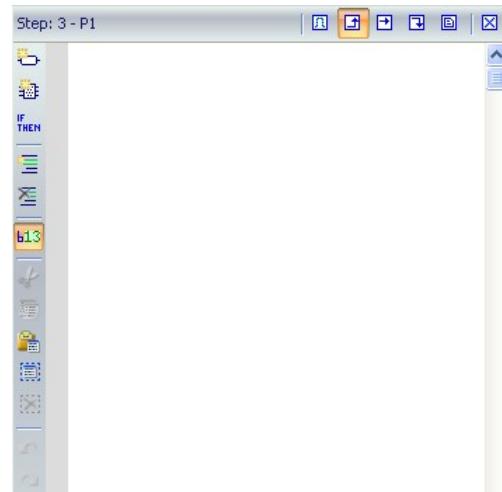
Select All



Delete selection

When editing P1 actions, use the Edit / Set Language menu command to select the programming language. This command is not available if the action block is not empty.

P1 actions than can be programmed in ST/IL text, LD or FBD.





Insert a new variable



Insert a new FB



Opens a list of keywords to use (applicable for ST language)



Insert a comment to an existing ST line



Remove a comment from an existing ST line



????????



Cut



Copy



Paste



Select All



Delete selection



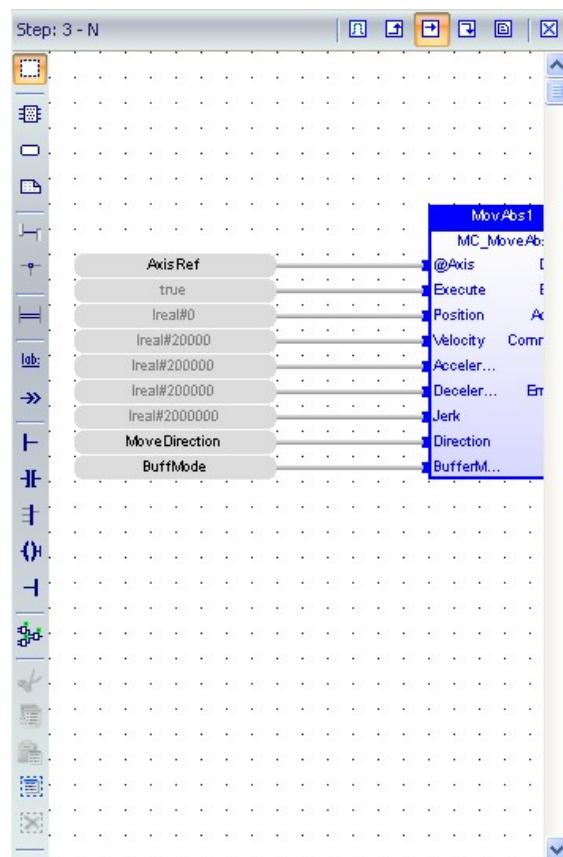
Undo



Redo

When editing N actions, use the Edit / Set Language menu command to select the programming language. This command is not available if the action block is not empty.

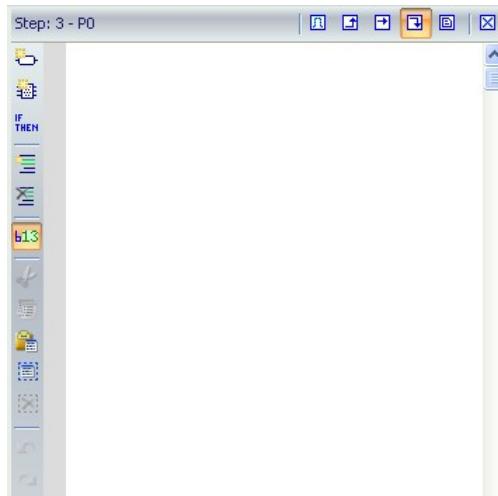
N actions than can be programmed in ST/IL text, LD or FBD





When editing P0 actions, use the Edit / Set Language menu command to select the programming language. This command is not available if the action block is not empty.

P0 actions than can be programmed in ST/IL text, LD or FBD.



Text notes



The first view ("Action") contains all simple actions to control a Boolean variable or a child SFC chart. However, it is possible to directly enter action blocks programmed in ST together with other actions in this view. Use the following syntax for entering ST action blocks in the first pane:

ACTION (qualifier) :

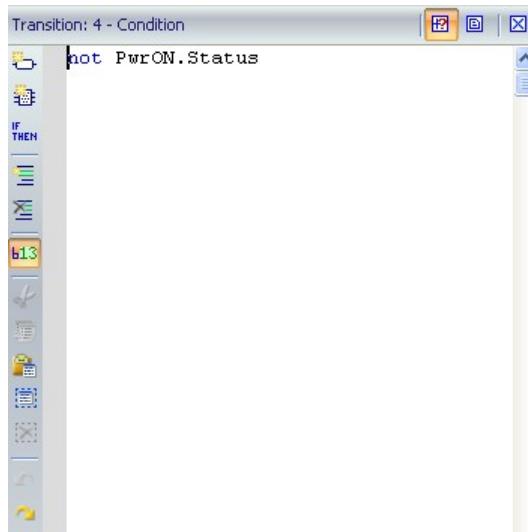
statements...

END_ACTION;

Where the qualifier is "P1", "N" or "P0".

1.9.1.6. Entering a transition condition

The condition and notes attached to a transition (level 2) are entered in a separate window. To open the level 2 editing window of a step or transition, double click on its symbol in the chart, or select it and press <Enter>.



The level 2 editing window consists of two views for entering various types of level 2 Information:

- Condition programmed in ST/IL text or LD
- Text notes

When editing the condition, use the Edit / Set Language menu command to select the programming language. This command is not available if the condition is not empty. FBD cannot be used to program a condition.

1.9.1.7. Entering step and transition notes

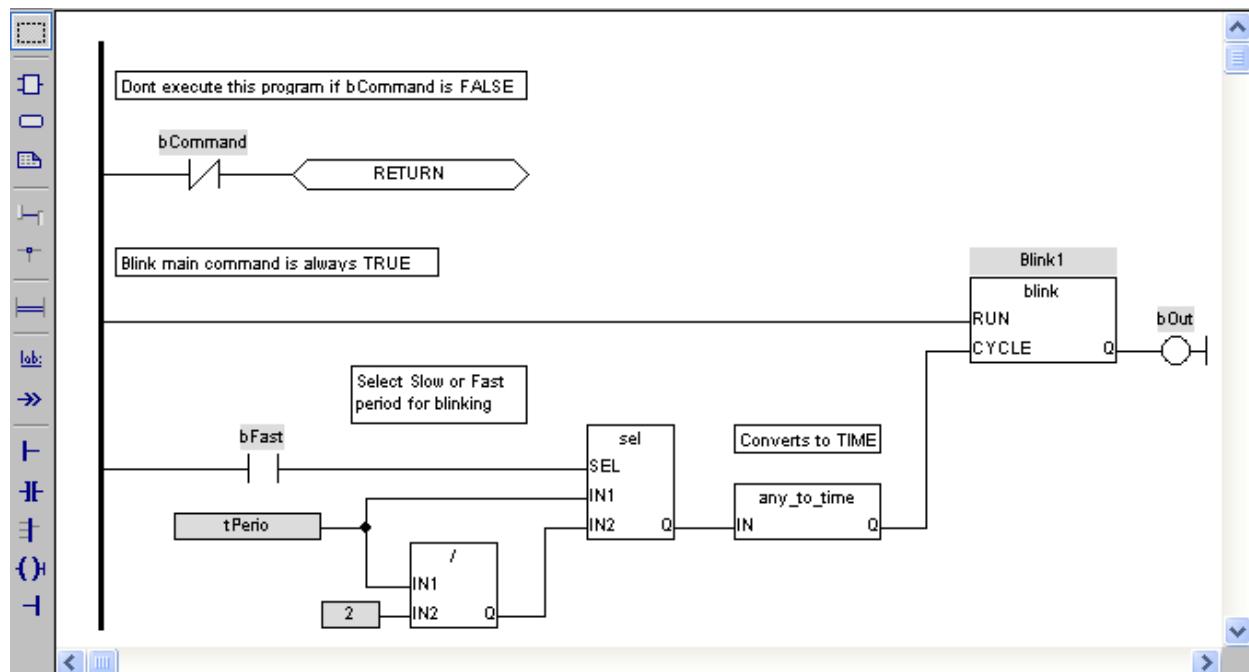
The SFC editor supports the definition of text notes for each step and transition. The notes are entered in the level 2 editing window of steps and transitions. Refer to the following topics for further information about the level 2 editing window:

- entering Level 2 for steps
- entering Level 2 for transitions

Notes can be displayed in the chart. Notes have no meaning for the execution of the chart. Entering notes for steps and transitions enables you to enhance the auto-documentation of your programs. It also provides an easy way to write and exchange specifications of an SFC program before actions and conditions are programmed.

1.9.2. Function Block Diagram (FBD) Editor

The FBD editor is a graphical tool that enables you to enter and manage Function Block Diagrams according to the IEC 61131-3 standard. The editor supports advanced graphic features such as drag and drop, object resizing and connection lines routing features, to rapidly and freely arrange the elements of your diagram. It also enables you to insert graphic elements in a FBD diagram of the LD (Ladder Diagram) language such as contacts and coils.



FBD diagram components

- Function blocks
- Variable tags
- Comment texts
- Corners
- Network breaks
- Labels
- Jumps
- Use of ST instructions

LD components

- Contacts
- Coils
- "OR" vertical rail
- Power rails

Related sections

- Using the FBD toolbar
- Selecting function blocks
- Drawing connection lines
- Selecting and entering variables and FB instances
- Viewing the diagram
- Moving or copying parts of the diagram
- Inserting an object on a line
- Resizing objects
- Bookmarks



Info: When a contact or a coil is selected, You can press the SPACE bar to change its type (normal, negated, pulse...).



1.9.2.1. Using the FBD toolbar

The vertical toolbar on the left side of the editor contains buttons for all available editing features. Click the required button before using the mouse in the graphic area.

Icon	Description
	Selection: When this icon is active, elements can be inserted in the diagram. Use the mouse to select object, lines, and tag name areas, or move, copy objects in the diagram. At any time, you can press the ESCAPE key to go back to the Selection mode.
	In this mode, elements cannot be inserted in the diagram.
	Function Block: Insert a function block to the diagram. Click in the diagram and drag the new function block to the desired position. The type of function block inserted is the one currently selected in the list of the main toolbar.
	Insert variable: Use the mouse to insert variable tags. Variable tags can then be wired to the input and output pins of the blocks. Click in the diagram and drag the new variable to the desired position.
	Insert comment text: Use the mouse to insert comment text areas in the diagram. Comment texts can be entered anywhere. Click in the diagram and drag the text block to the desired position. The text area can then be selected and resized.
	Insert connection line: Use the mouse to wire input and output pins of the diagram objects. The line must always be drawn in the direction of the data flow: from an output pin to an input pin. The FBD editor automatically selects the best routing for the new line. You can change the default routing by inserting corners on lines. (see below) You also can drag a line from an output pin to an empty space. In that case the editor automatically finished the line with a user defined corner so that you can continue drawing the connection to the desired pin and force the routing while you are drawing the line.
	Insert corner: In this mode, use the mouse for inserting a user defined corner on a line. Corners are used to force the routing of connection lines, as the FBD editor imposes a default routing only between two pins or user defined corners. Corners can then be selected and moved to change the routing of existing lines.
	Insert network break: In this mode, use the mouse for inserting a horizontal line that acts as a break in the diagram. Breaks have no meaning for the execution of the program. They just help the understanding of big diagrams, by splitting them in a list of networks.
	Insert label: Use the mouse to insert a label in the diagram. A label is used as a destination for jump symbols (see below).
	Insert jump: Use the mouse to insert jump symbols in the diagram. A jump indicates that the execution must be directed to the corresponding label (having the same name as the jump symbol). Jumps are conditional instructions. They must be linked on their left side to a Boolean data flow.



Insert left power rail: Use the mouse to insert a left power rail in the diagram. A left power rail is an element of the LD language, and represents a *TRUE* state that can be used to initiate a data flow. Power rails can then be selected and resized vertically according to the desired network height.



Insert contact: Use the mouse to insert a contact in the diagram as in Ladder Diagrams.



Insert “OR” rail: Use the mouse to insert a rail that collects several Boolean data flows for an “OR” operation, in order to insert parallel contacts such as done in Ladder Diagrams.



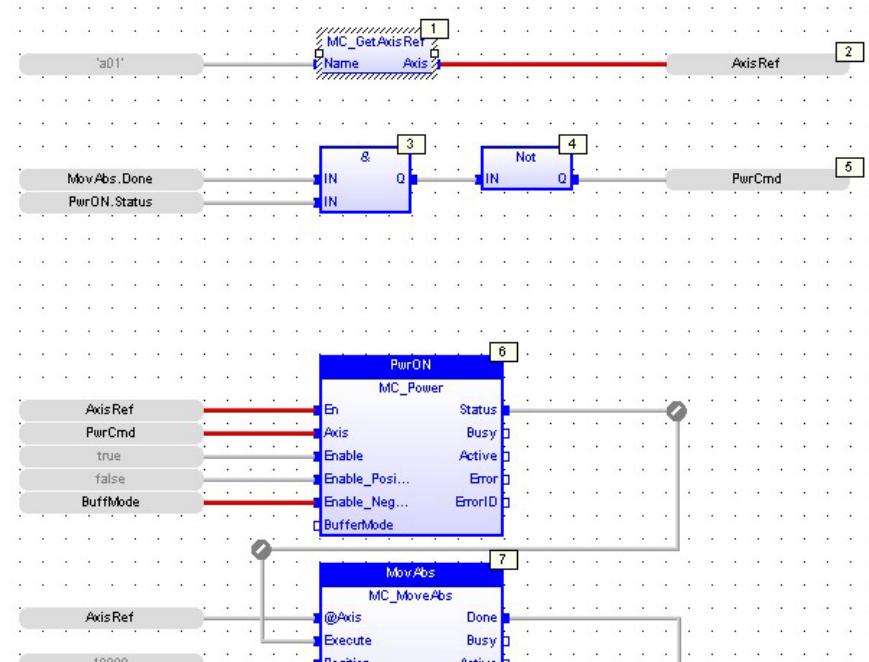
Insert coil: Use the mouse to insert a coil in the diagram as in Ladder Diagrams. It is not mandatory that a coil be connected on its right side.



Insert right power rail: Use the mouse to insert a right power rail in the diagram. A right power rail is an element of the LD language, and is commonly used for terminating Boolean data flows. However it is not mandatory to connect coils to power rails. Right power rails have no meaning for the execution of the diagram.



Display Execution Order: when selected, displays the execution order of the FBD.



Cut



Copy



Paste



Select All



Delete selection



Undo



Redo



1.9.2.2. FBD variables

All variable symbols and constant expressions are entered in FBD diagrams using small boxes. Click to insert a variable tag:

Icon	Description
	Insert variable: Use the mouse to insert variable tags. Click in the diagram and drag the new variable to the desired position.

Double click on a variable tag to open the variable selection box and either select the symbol of the desired variable or enter a constant expression.

Variables tags must then be linked to other objects, such as block inputs and outputs using connection lines.

You can resize a variable box vertically in order to display its tag (short comment text), its description text, and its I/O location, together with the variable name, if the variable is mapped to an I/O channel. The variable name is always displayed at the bottom of the rectangle:

tag	% location
description	name

1.9.2.3. FBD comments

Comment text areas can be entered anywhere in a FDB diagram. Click to insert a new comment area:

Icon	Description
	Insert comment text: Use the mouse to insert comment text areas in the diagram. Comment texts can be entered anywhere. Click in the diagram and drag the text block to the desired position.

Double click on the comment area for entering or changing the attached text. When selected, comment texts can be resized.

1.9.2.4. FBD corners

Corners are used to force the routing of connection lines, as the FBD editor imposes a default routing only between two pins or user defined corners. All variable symbols and constant expressions are entered in FBD diagrams using small boxes. Click to insert a corner on a line:

Icon	Description
	Insert corner: Use the mouse to insert a user defined corner on a line.

You can drag a new line from an output pin to an empty space. The editor then automatically completes the line with a user defined corner so that you can continue drawing the connection to the desired pin and force the routing while you are drawing the line.

Corners can then be selected and moved to change the routing of existing lines.



1.9.2.5. FBD network breaks

Network breaks can be entered anywhere in an FBD diagram. Breaks have no meaning for the execution of the program. They simply help in the understanding of large diagrams, by splitting them into a list of networks. Click  to insert a new break:

Icon	Description
	Insert network break: Use the mouse to insert a horizontal line that acts as a break in the diagram.

The line break is drawn on the whole diagram width. No other object can overlap a network break. Line breaks can then be selected and moved vertically to another location.

Network breaks can also be used for browsing the diagram. Press **<Ctrl>+<Page Up>** or **<Ctrl>+<Page Down>** keys to move the selection to the next or previous network break.

1.9.2.6. FBD "OR" vertical rail

The FBD editor enables the drawing of LD rungs. Either a specific object or the "OR" rail can be inserted on a rung in order to connect parallel contacts together. Click  to insert a new "OR" rail:

Icon	Description
	Insert OR rail: Use the mouse to insert a rail that collects several Boolean data flows for an OR operation. This is performed so that parallel contacts can be inserted as is done in Ladder Diagrams.

The OR rail has exactly the same meaning as an OR block regarding the execution of the diagram.

1.9.2.7. Drawing FBD connection lines

Icon	Description
	Click this icon before inserting a new line.

The editor enables you to terminate a connection line with a Boolean negation represented by a small circle. To set or remove the Boolean negation, select the line and press the SPACE bar.

Connection lines must always be drawn in the direction of the data flow, from an output pin to an input pin. The FBD editor automatically selects the best routing for the new line. Connection lines indicate a data flow between the following possible objects:

Icon	Description
	Function Block: Refer to the help on the function block for the description of its input and output pins, and the expected data types for the coherency of the diagram.
	Variable: Variable can be connected on their right side (to initiate a flow) or on their left side for forcing the variable, if it is not "read only". The flow must fit the data type of the variable.
	Jump: A jump must be connected on its left side to a Boolean data flow.
	Left power rail: Left power rails represent a <i>TRUE</i> state and can be connected to a non



Icon	Description
	limited number of objects on their right side.
	Contact: A contact must be connected on its left side and on its right side to Boolean data flows.
	“OR” rail: Such rail that collects several Boolean data flows for an “OR” operation, in order to insert parallel contacts such as done in Ladder Diagrams. It may have several connections on its left side and on its right side. All connected data flows must be Boolean.
	Coil: A coil must be connected on its left side to a Boolean data flow. It is not mandatory that a coil be connected on its right side.
	Right power rail: A right power rail is an element of the LD language, and is commonly used for terminating Boolean data flows. It has a non limited number of connections on its left side. It is not mandatory to connect coils to power rails.

1.9.2.8. Selecting FBD variables and instances

Icon	Description
	Press this button or press ESCAPE before any selection.

To select the name of the declared variable to be attached to a graphic symbol, you must be in *Selection* mode. Simply double click on the tag name gray area. The following types of object must be linked to valid symbols:

Icon	Description
	Function Block: If it is a function block, you must specify the name of a valid declared instance of the corresponding type.
	Variable: Must be attached to a declared variable. Alternatively, a variable box may contain the text of a valid constant expression.
	Label: Must have a name. The name must be unique within the diagram.
	Jump: must have the same name as its destination label.
	Contact: Must be attached to a declared Boolean variable.
	Coil: Must be attached to a declared Boolean variable.

Symbols of variables and instances are selected using the variable list, that can be used as the variable editor. You can simply enter a symbol or constant expression in the edit box and press **OK**. You also can select a name in the list of declared object, or declare a new variable by pressing the **Create** button.



1.9.2.9. Viewing FBD diagrams

The diagram is entered in a logical grid. All objects are snapped to the grid. You can use the commands of the **View** menu for displaying or hiding the points of the grid. The (x,y) coordinates of the mouse cursor are displayed in the status bar. This helps you locating errors detected by the compiler, or aligning objects in the diagram.

At any time you can use the commands of the **View** menu for zooming in or out the edited diagram. You also can press the [+] and [-] keys of the numerical keypad for zooming the diagram in or out.

1.9.2.10. Moving or copying FBD objects

Icon Description

Press this button or press ESCAPE before selecting objects.

The FBD editor fully supports drag and drop for moving or copying objects. To move objects, select them and simply drag them to the desired position.

To copy objects, you may do the same, and just press the CONTROL key while dragging. It is also possible to drag pieces of diagrams from a program to another if both are open and visible on the screen.

At any moment while dragging objects you can press ESCAPE to cancel the operation.

Alternatively, you can use classical **Copy / Cut / Paste** commands from the Edit menu. When you run the Paste command, the editors turns in *Paste* mode, with a special mouse cursor. Click in the diagram and move the mouse cursor to the desired position for inserting pasted objects.

1.9.2.10.a Using the keyboard

When graphic objects are selected, you can move them in the diagram by hitting the following keys:

Shortcut	Description
Shift + Up	Move to the top.
Shift + Down	Move to the bottom.
Shift + Left	Move to left.
Shift + Right	Move to right.

When an object is selected, you can extend the selection by hitting the following keys:

Shortcut	Description
Shift + Control + Home	Extend to the top: select all objects before the selected one.
Shift + Control + End	Extend to the bottom: select all objects after the selected one.

To insert or delete space in the diagram, you can simply select an object, press Shift+Control+End to extend the selection and then move selected objects up or down.



1.9.2.10.b Auto alignment

When objects are selected, the following keystrokes automatically align them:

Shortcut	Description
Control + Up	To the top.
Control + Down	To the bottom.
Control + Left	To left.
Control + Right	To right.

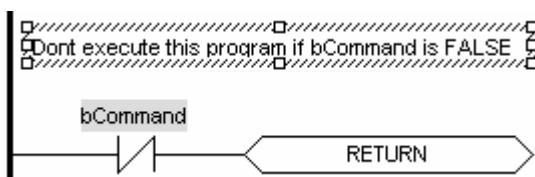
1.9.2.11. Inserting FBD objects on a line

The FBD editor allows you to insert an object on an existing line and automatically connect it to the line. This feature is available for all objects having one input pin and one output pin, such as variable boxes, contacts and coils. This feature is mainly useful when entering pieces of Ladder Diagrams. Just draw a horizontal line between left and right power rails; this is the rung, then insert contacts and coils on the line to build the LD rung.

1.9.2.12. Resizing FBD objects

Icon	Description
	Press this button or press ESCAPE before selecting objects

When an object is selected, small rectangles indicate how to resize it. Click on the small rectangles to resize the object in the desired direction.



Not all objects can be resized. The following table indicates possible operations:

Item	Possible operations
Variable	Horizontally and vertically. Resizing a variable box vertically enables you to display the variable name, its tag (short comment text), its description text, plus its I/O location if the variable is mapped to an I/O channel, together. The variable name is always displayed at the bottom of the rectangle: % location description tag name
Block	Horizontally

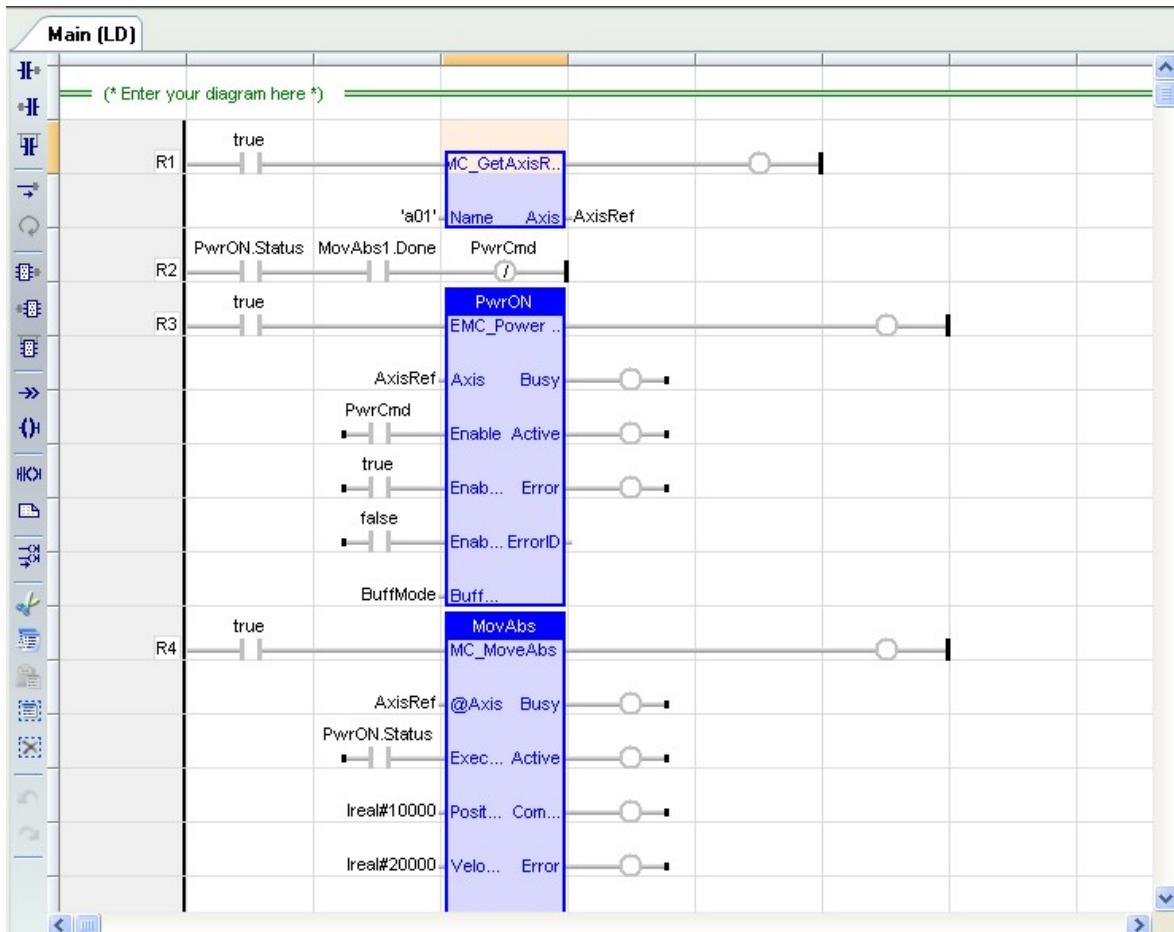


Labels and jumps	Horizontally
Power rails	Vertically
OR rail	Vertically
Comment area	In all directions



1.9.3. Ladder Diagram (LD) Editor

The LD editor is a graphical tool enabling entry and management of Ladder Diagrams. The editor allows quick input using the keyboard, and supports advanced graphic features such as drag and drop.



LD diagram components	Related sections
Rungs	Using the LD toolbar
Contacts	Selecting function blocks
Coils	Selecting and entering variables and FB instances
Power rails	Viewing the diagram
Function blocks	Moving or copying parts of the diagram
Labels	Bookmarks
Jumps	
Comments	
Use of ST instructions	



Info
When a contact or a coil is selected, change its type (normal, negated, pulse...) by pressing the <Space> bar.



1.9.3.1. Using the LD toolbar

The vertical toolbar on the left side of the editor contains buttons for inserting items in the diagrams. Items are inserted at the current position in the diagram.

Icon, Shortcut	Description
Shift+F4	Insert a contact before the selected item
F4	Insert a contact after the selected item
Ctrl+F4	Insert a contact in parallel with the selected items
Ctrl+Space	Insert a horizontal line before the selected item so that it is pushed to the right
<Space>	Replace the item style
Shift+F8	Insert a function block before the selected item
F8	Insert a block after the selected item
Ctrl+F8	Insert a block in parallel with the selected items
Shift+F9	Add a jump in parallel with the selected coil
F9	Add a coil in parallel with the selected coil
Ctrl+R	Insert a new rung in the diagram
Ctrl+D	Insert a comment between rungs
	Align the coils
	Cut
	Copy
	Paste
	Select All
	Delete selection
	Undo
	Redo



1.9.3.2. Managing Rungs

Icon Description

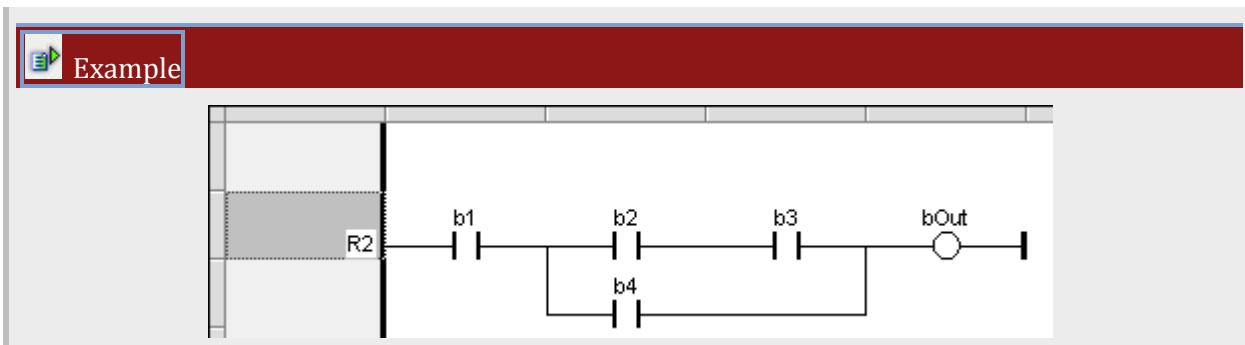


Click this icon in the LD toolbar to insert a new rung

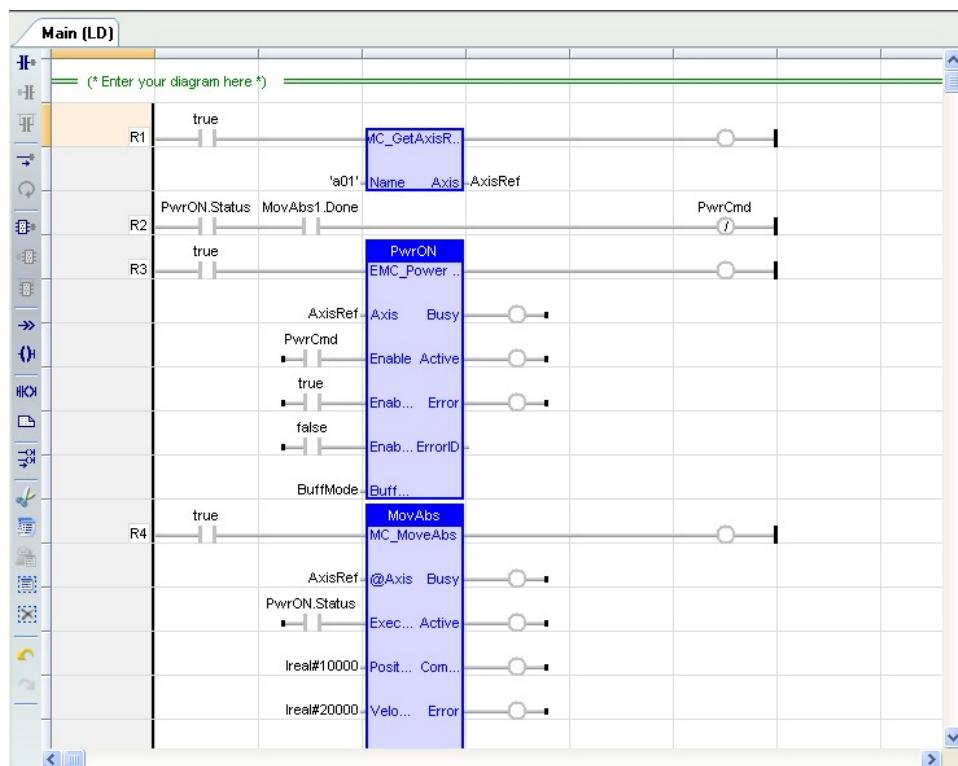
A LD diagram is a sequential list of rungs. Each rung represents left to right Boolean power flow, that begins with a power rail, always drawn in the first column of the diagram, and ends with a coil or a jump symbol.

Each Rung is identified by a default numbered identifier (**Rnnn**) displayed on the left of the power rail. The rung identifier can be used as a target for jump instructions. Alternatively a specific rung label can be entered by double clicking in the rung head in the left margin.

The LD editor enables you to manipulate whole rungs by selecting only their head in the left margin. The following Example shows a selected rung:



When a rung is selected, the commands delete, copy or cut are accessible.





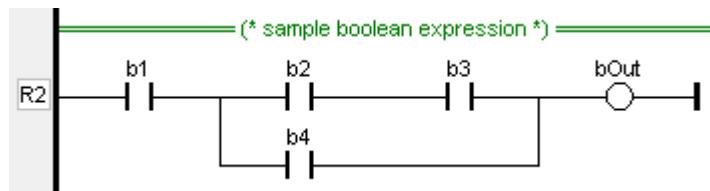
1.9.3.3. Comments in LD diagrams

Icon Description



Click this icon in the LD toolbar to insert a new comment line.

The LD editor enables you to insert comment texts in the diagram. A comment is a single line of text inserted between two rungs. The comment text is displayed on a double line in the diagram:

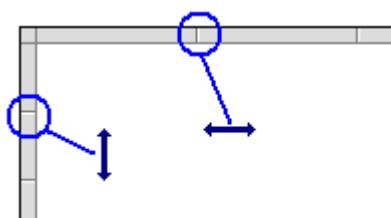


Comment texts have no meaning for the execution of the diagram. They are used to enhance the readability of the program, enabling the description of each rung.

The comment text remains visible when the diagram is scrolled horizontally. To change the text of the comment, place the selection anywhere on the comment line and hit <Enter> key, or simply double click on the comment line.

1.9.3.4. Viewing LD diagrams

The diagram is entered in a logical grid. All objects are snapped to the grid. You can drag the separation lines in vertical and horizontal rulers to freely resize the cells of the grid:



The LD editor adjust the size of the font according to the zoom ratio so that the name of variables associated with contacts and coils are always visible. If cells have sufficient height, variable names are completed with other pieces of information about the variable:

- Its tag (short description).
- Its description text.
- Its I/O name (%...) if the variable has a user defined name.

1.9.3.5. Moving and copying LD items

The LD editor fully supports drag and drop for moving or copying objects. To move objects, select them and simply drag them to the desired position, in the same rung or in another rung.

To copy objects, you may do the same, and just press the <Ctrl> key while dragging. It is also possible to drag pieces of diagrams from a program to another if both are open and visible on the screen.

At any moment while dragging objects you can press <Esc> to cancel the operation.

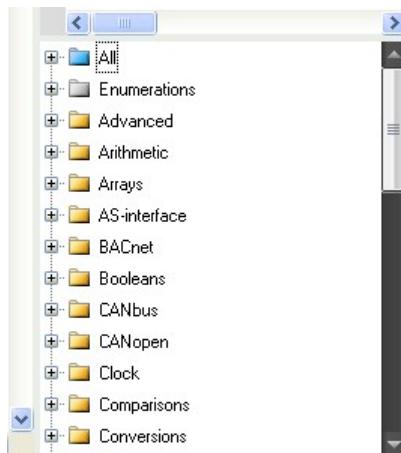
Alternatively, you can use classical **Copy / Cut / Paste** commands. Paste is performed at the current position.



You can manipulate whole rungs by selecting only their head in the left margin (select only the cell where the rung number is displayed).



1.9.4. Selecting function blocks



All available operators, functions and function blocks are listed in the bottom/right area of the editor. The list of available blocks is sorted into categories.

The All category displays the complete list of available blocks. The Recent category contains the last used blocks. The Project category lists all UDFBs and sub-programs declared in the project.

To insert a function block in a program, simply select it in the list and drag it with the mouse to the desired position in the edited text or diagram.

Press the **F1** key when a function block is selected to obtain help about a function's input and output pins. In selection mode, you also can double-click the mouse in a function block of the diagram to change its type, and set the number of input pins if the function block can be extended.



1.9.5. Selecting variables and instances

Name	Type	Dim.
Global variables		
RETAIN variables		
Main		
PwrON	MC_Power	
MovAbs	MC_Move...	
PwrCmd	BOOL	
BuffMode	MC_BUFF...	
MoveDirecti...	MC_DIRE...	
AxisRef	AXIS_REF	
MovAbs1	MC_Move...	

Symbols of variables and instances are selected using the variable list, that can be used as the variable editor. Selecting variables is available from all editors:

- In FBD diagrams, double click on a variable box, a FB instance name, a contact or a coil to select the associated variable.
- In LD diagrams, double click on a contact, a coil or a function block input or output to select the variable. Double click on the top of a FB rectangle to select an instance.

When the variable editor is visible in the editor window, it is possible to simply drag a variable from the list to the program and insert it.

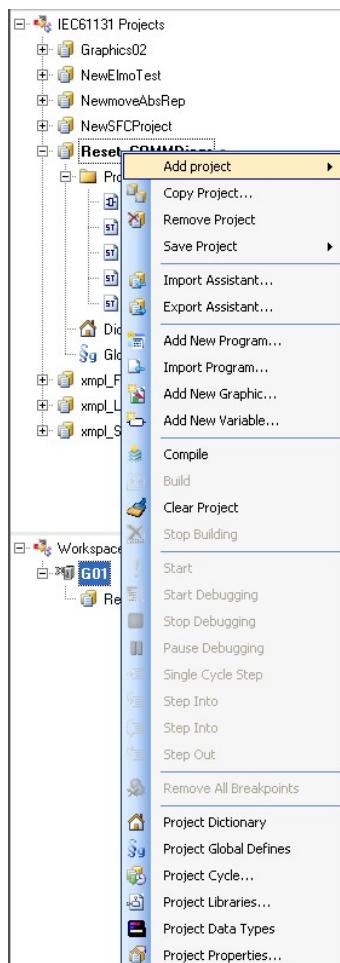
When inserting a variable name manually, the Workbench automatically checks if it is present and if not, suggests that it should be declared before insertion.



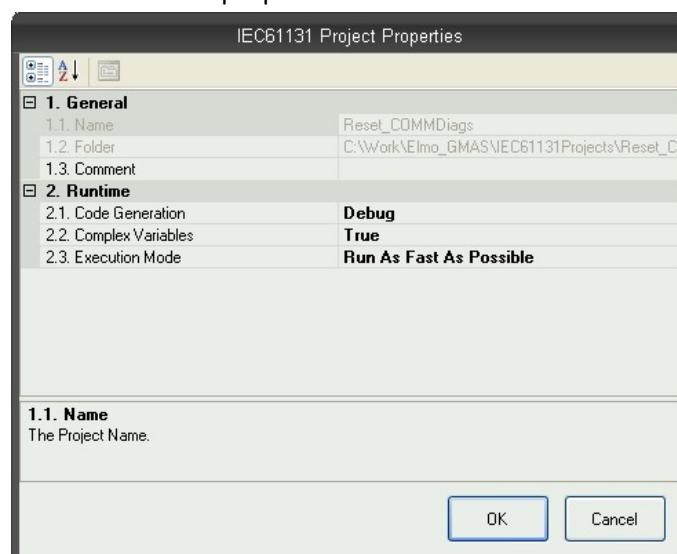
1.10. Project Settings

To set a Projects properties:

1. Right-click on the Project, and select **Project Properties** from the menu. The IEC61131 Project Properties window opens.



2. Select to change from the Runtime properties.

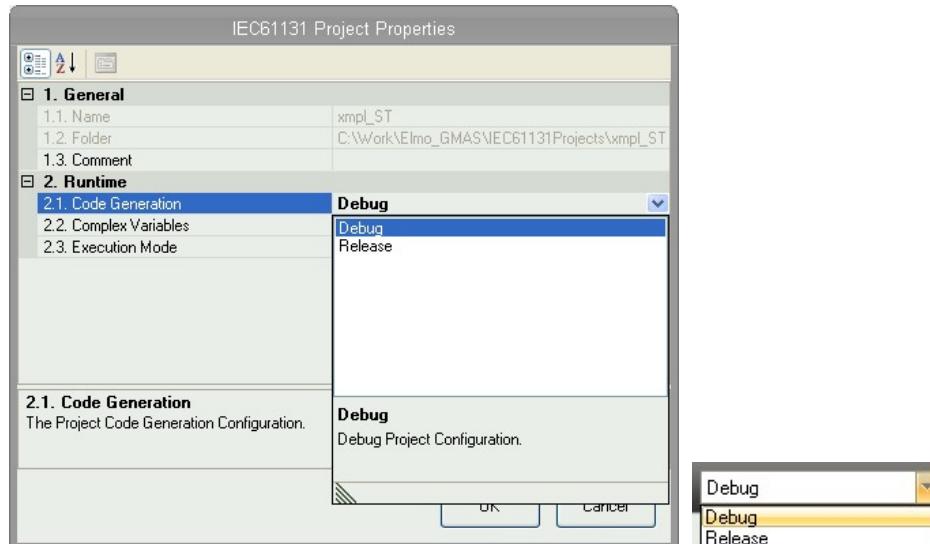




1.10.1. Project Settings Panel

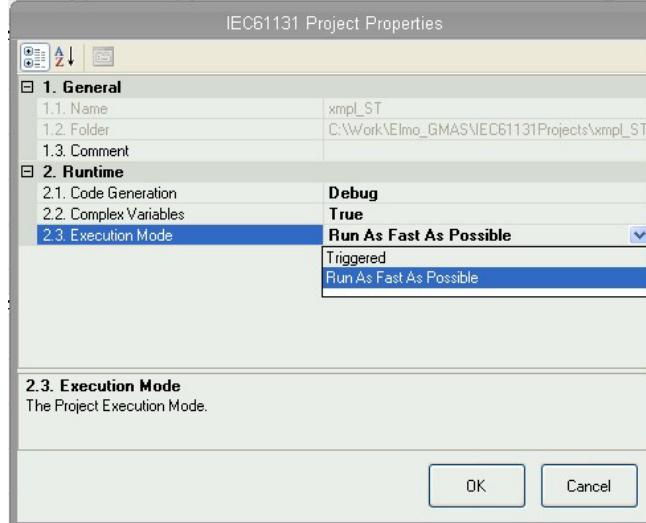
This panel enables you to define main parameters for building and running the runtime application.

Code generation



For step-by-step debugging to be available during simulation or On Line testing, select the DEBUG compiling mode. If step by step debugging is no more required, select the RELEASE compiling mode in order to give the highest performance to the runtime application.

Execution mode



You must specify in this box how the cycles must be triggered at run-time:

- **Run As Fast As Possible**

The target does not wait between two cycles. The target simply runs as fast as possible

- **Triggered**

The duration of a cycle must be specified, expressed as a number of microseconds. Refer to OEM instructions for explanations of the supported accuracy for the cycle timing of your target system.



External objects

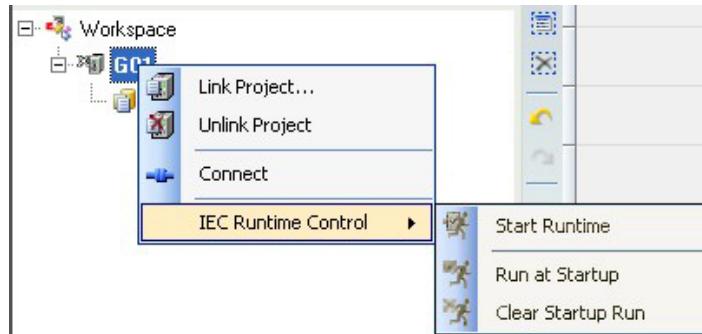
The workbench allows defined objects from other projects, such as UDFBs, to be used in the present project. This provides an easy way to manage libraries of IEC defined function blocks that can be re-used in various projects. In addition to this feature, the workbench provides an easy way to safely handle externally defined objects in the local project. Refer to the section 1.19 Libraries on page 149 for more details.



1.10.2. Target Panel

This panel enables you to select the kind of runtime installed in your target system.

Target



You must specify the type of runtime installed in your target system. The IEC Runtime Control has the following options:

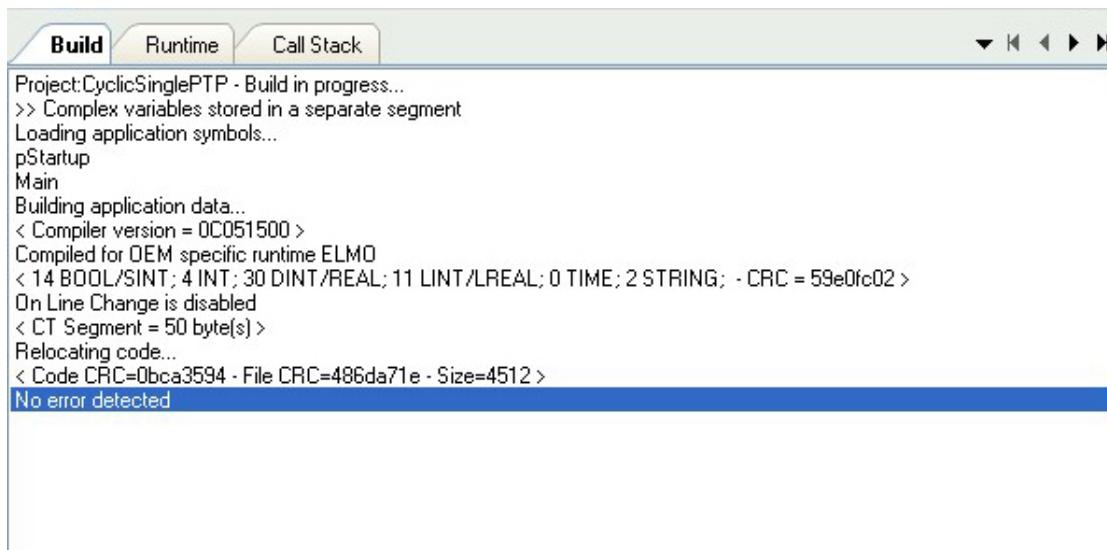
- Start Runtime
- Run at Startup
- Clear Startup Run

Identification

A target is identified by a name. This name will be included in the built code and checked at runtime. In addition, you must specify the suffix of the file generated at build time that contains the downloaded code. Download is not possible if the suffix entered here does not match the one expected by the runtime.



1.10.3. Build and Runtime Tabs



These tabs provide information of the processes operation during specific routines e.g. Building, Running, etc.

Code generation

If you want step by step debugging to be available during simulation or On Line test, you need to select the DEBUG compiling mode. If step by step debugging is no more required, you should select the RELEASE compiling mode in order to give highest performance to the runtime application.

Compiling Options

The following options are available during compiling:

Enable FBD line coloring during debug

- Display warning messages (uncheck this option to remove all warning reports during build)
- Enable large jump instructions
For very large programs, jump instructions (e.g. for IF and loops ST instructions) may lead to compiling errors because the target of the jump is too far. In that case you need to check this option.
- Embed all symbols with the downloaded application code
- Keep the case of embedded symbols (if not checked, all embedded symbols are turned uppercase).

Check SFC chart safety

Check array bounds at runtime

- Remove the code of unreferenced sub-programs and UDFBs
If you check this option, the code of unused sub-programs and UDFBs is not included in the compiled code. This enables you to reduce the size of the downloaded code. Note that such sub-programs and UDFBs still exist in the code and their variables are defined, so that possible later On Line Change remains possible.
- Check IEC conformity
If this option is checked, a number of warning messages are output every



time you use in your program specific functions or function blocks that are not defined in the IEC standard. This only applies to embedded function blocks and not to UDFBs written in IEC language.

- Allocate status bits for variables with embedded properties

Runtime password Optionally you can define a password for the target application. If the password is defined in the compiling option, the user will be prompt for entering the password each time a remote connection is established to the target. The password is a number. If the password is "0", it means that no password is defined.

Coloring FBD Flow lines As an option, the debugger may display real time status of FBD flow lines during debug of simulation.

This option must be selected before compiling the project. It is selected from the **Enhanced** tab of the **Compiling Options** dialog box.

When this option is selected, the compiler automatically adds variables and instructions to the generated application code so that status of all FBD flow lines can be viewed during debug of the simulation.

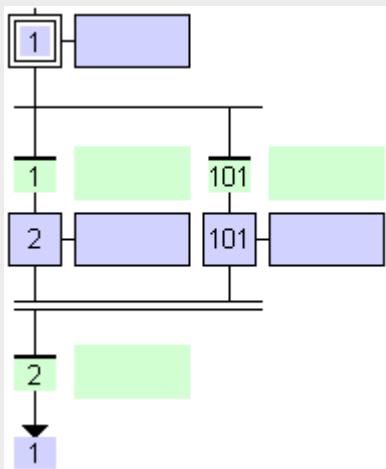
Selecting this option increase the size of the downloaded code and may lead to less runtime performance. It is recommended to uncheck this option when the application is fully tested and compiled in RELEASE mode for its final implementation.

Safety build options The compiler and debugger includes several options that enable you to increase the safety of your runtime applications. Such options are selected from the **Enhanced** tab of the **Compiling Options** dialog box:

Check array bounds at runtime When this option, is checked, the compiler automatically include extra instructions in the application code so that bounds are checked before reading or writing any array item. It is highly recommended to check this option during the development and validation of your applications.

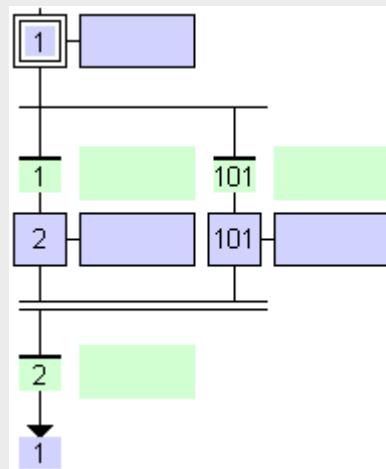
Unchecking this option allows to decrease the application code size and leads to higher runtime performance.

Check safety of SFC charts As an option, the compiler may check the safety of all SFC charts at compiling time. This enables the detection of possible unsafe or blocking situations due to the form of the SFC chart. Below are some Examples of such situations.

**Example**

Possible unsafe situation:

As there is no "AND" convergence, several steps (such as 1 and 2) may be active at the same time.



Possible blocking situation:

As either transition 1 or 101 is proceeded, the transition 2 can never be proceeded.

Info:

The compiler provides warning messages when checking SFC charts and no more.

1.10.3.1. Management of complex variables at runtime

The following option in the Enhanced tab of the compiling options dialog box defines how complex variables are managed at runtime:

Check Description

- Store complex variables in a separate segment.

If this option is set, then all complex variables are stored separately from other variables in the memory of the T5 runtime. If not, complex variables are stored together with single variables.

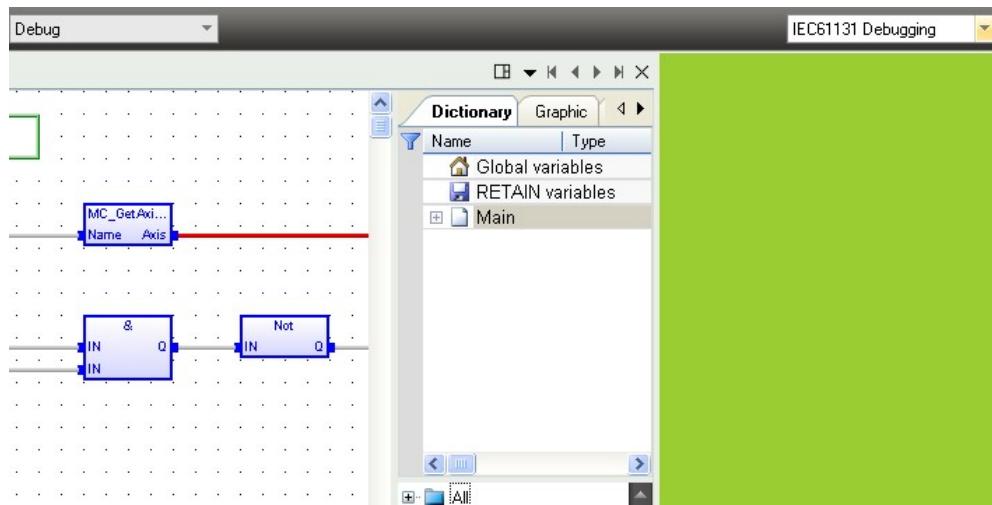
This option must be used carefully. Setting up this option enables more programming possibility regarding complex variables and extends the capacity of the runtime memory. On another hand, the runtime application may have lower performance when this option is set. It is recommended to keep this option off as far as you do not need extra features related to it. This ensures that your application has highest runtime performances.

Complex variables are arrays, structures, and instances of function blocks. When this option is set, the following features are allowed for programming:

- You can use arrays of structures.
- You can use arrays of FB instances.
- You can pass any complex data (array, structure, instance) to a UDFB or sub-program.
- There is nearly no limitation in the amount of complex data declared.
(theoretically up to 4GB - but practically limited by the memory available in the runtime)



1.10.4. Debug window



This window enables you to select the main options applied when simulating or debugging the runtime application.

Simulation If you select the *Always start simulator in Cold Start mode*, then the box normally opened when running the simulator for starting mode selection is not displayed.

This option applies to any project.

Connected to the runtime These options enable you to impose some prompt messages for a user confirmation in most sensitive debug commands:

- Stop the runtime application.
- Download application code.
- On Line change.

These options apply to any project.

When starting the runtime These options allow you to select which kind of starting mode should be proposed to the user per default when starting the runtime application. Note that the user will still be allowed to change the starting mode in the box.

Communication parameters Here you can select the way communication parameters are set:

- Use current settings: check this option for always starting communication with the latest selected parameters.
- Always use: alternatively, you can enter here the communication parameters that should be used for this project.



1.10.5. Project On Line Changes

The workbench allows certain changes to an application "on the fly" while it is running in the target system. The following changes can be performed:

- Variable changes.
- Change the condition of a SFC transition or the actions of a SFC step.
- Create, rename or delete global and local variables.

The following changes are not allowed:

- Create, delete or rename a program.
- Change SFC charts.
- Change the local parameters and variables of a UDFB.
- Change the type or dimension (or string length) of a variable or function block instance.
- Change the set of I/O boards.
- Change the definition of RETAIN variables.

In addition, the following programming features that are not safe during a change are forbidden:

- Pulse (P or N) contacts and coils (edge detection). Instead, you must use declared instances of R_TRIG and F_TRIG function blocks.
- Loops in FBD with no declared variable linked. You need to explicitly insert a variable in the loop.

On Line Change will be denied by the runtime if some system resources are currently open when the change is performed. A system resource can be either a file open by file management functions, or a socket open by TCP/UDP functions, or an dynamic array created by ARCREATE functions.

In order to allow the declaration of new variables and function blocks, you have to define the amount of memory to be allocated in the target for each type of data. This includes:

- The number of variable for each type (8, 16, 32 or 64 variables, character strings).
- The number of function block instances.
- The amount of memory for storing character strings.
- The amount of memory for private data of function block instances.

The number of variables of each type actually used in the application is given as a report at the end of each build. Changes are allowed until you exceed the sizing for at least one type of data. At this time, if you need to declare new variables, you have to change the configuration of memory sizing, rebuild the application and perform a full download stopping the target application.



When On Line Change is enabled, all new created variables are marked in blue in the variable editor. Deleted variables are kept as *ghosts*, marked in gray and renamed with a "*_del_*" prefix. You are allowed to manually rename a ghost variable if you want to make it alive again.



The On Line Change feature can be experimented with the Simulator started in *Hot Restart* mode. A hot restart is possible when On Line Change is enabled and when the application includes valid changes.



1.11. Definitions

The compiler supports the definition of aliases. An alias is a unique identifier that can be used in programs to replace another text. Definitions are typically used for replacing a constant expression and ease the maintenance of programs.

There are three levels of definitions:

- Common to all the projects installed on your machine.
- Global to all programs of a project.
- Local to one program.

Definitions are entered in a text editor. Each definition must be entered on one line of text according to the following syntax:

```
#define Identifier Equivalence (* comments * )
```

Example

```
#define OFF FALSE      (* redefinition of FALSE constant * )
#define PI 3.14        (* numerical constant * )
#define ALARM (bLevel > 100) (* complex expression * )
```

You can use a definition within the contents of another definition. The definition used in the other one must be declared first.

Example

```
#define PI 3.14
#define TWOPI (PI * 2.0)
```

Info:

A definition may be empty, for Example:

```
#define CONDITION
```

Defined word can be used for directing the conditional compiling directives.

You can enter **#define** lines directly in the source code of programs in IL or ST languages.

The use of definitions may disturb the program monitoring and make error reports more complex. It is recommended to restrict the use of definitions to simple expressions that have no risk to lead to a misunderstanding when reading or debugging a program.



1.11.1. System definitions

The following words are predefined by the compiler and can be used in the programs:

Word	Description
<u>__DATE__</u>	Date stamp of compiling expressed as a string constant expression Format is: Year/Month/Day-Hours-Minutes.
<u>__MACHINE__</u>	Name of the machine where compiling is run, expressed as a string constant expression.
<u>__USER__</u>	Name of the user where compiling is run, expressed as a string constant expression.



Example

The following ST program sets variables declared as STRING(255):

```
strDate := __DATE__;  
strMachine := __MACHINE__;  
strUser := __USER__;
```

Result:

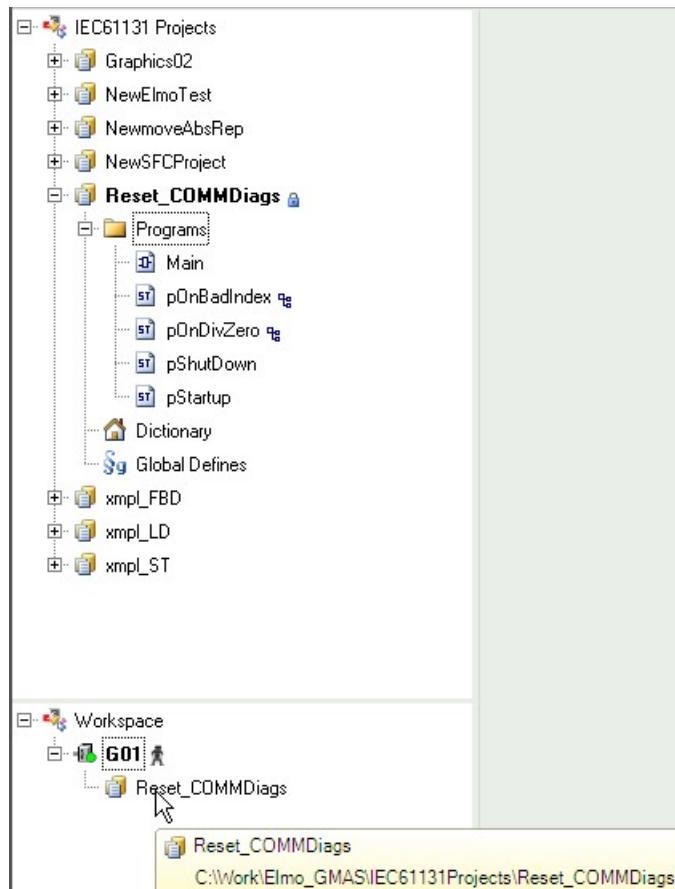
```
strDate is '2007/11/25-10:45'  
strMachine is 'LaptopJX'  
strUser is 'John'
```

1.12. Running the application

Runtime only applies to the project linked to the G-MAS, and can only occur with one project at a time.

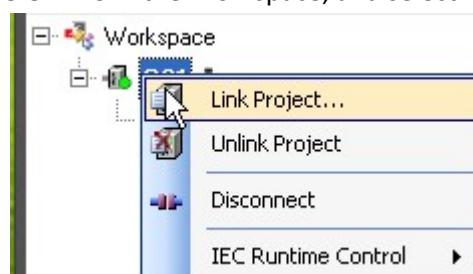
To select and link a project to the G-MAS:

1. Drag the highlighted project title to the G-MAS in the Workspace. The project is linked.



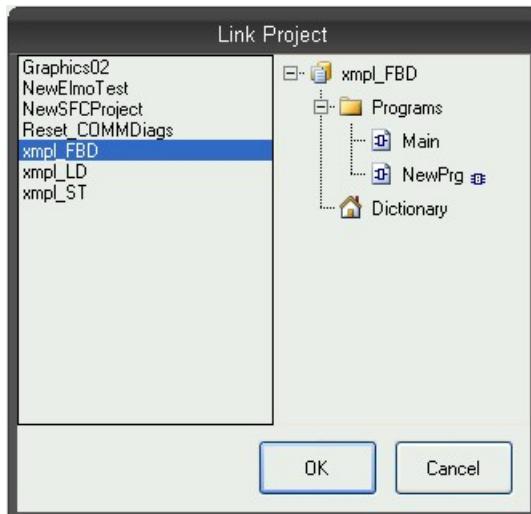
Alternatively,

- a. Right-click on the G-MAS in the Workspace, and select **Link Project**.





- b. From the available projects in the Link Project window, select a project to link to the G-MAS, and then click **OK**. The Project is linked.



From the program list, it is possible to control the execution of each program at run time. Use the command of the "Project" menu to:

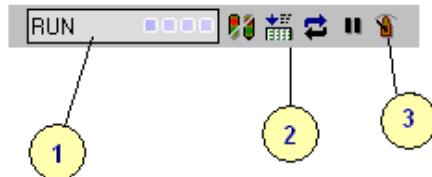
- Stop or start the selected program.
- Pause (suspend) or resume (restart) the selected program.

Stopped programs are displayed in red in the list of programs. Suspended programs are displayed in blue.



1.12.1. The control panel

When the workbench is used for test (On Line or Simulation), the toolbar provides information about run-time target and provides all commands for controlling the target:



1. Status of the target application.

Control commands.

Change the cycle triggering.

1.12.1.1. The application status

The status box provides global information about the target, the running application and the communication links. Possible states are:

State	Description
No connection	Connection is not established
Connecting	The workbench is establishing the communication
Disconnecting	The workbench is closing the connection
No application	Target is stopped
Other application	Another application runs on the target
Communication error	Communication error occurred
Bad version	The target runs a different version of the application
RUN	The application is running
STOP	The application is in cycle stepping mode
SFC break	The application is stopped on a SFC breakpoint
Error	The application is stopped on error

When an application is active in the target, you can move the mouse cursor on the state box to display the detailed versioning information in a tooltip.

1.12.1.2. Controlling the application, available commands

Icon	Description
	Starts the application.
	Stops the application.
	Pause the application in Cycle to Cycle mode.



Restart the application in normal mode.



Execute a single step.



Download: Loads the application if the target is stopped (no application).
Loads new code for On Line change if the application is running.



On Line change.



Step in (available during step by step debugging).



Step over (available during step by step debugging).



Step out (available during step by step debugging).

1.12.1.2.a Cycle timing

The cycle timing is displayed in a tooltip when you place the cursor upon the application status box. It includes:

- The last measured cycle time.
- The programmed cycle triggering period.
- The maximum (longest) detected cycle time.
- The number of timing overflow detected.



1.12.2. Using the editors in test mode

In Test (On Line or Simulation) mode, all editors are animated with real time values of the edited objects:

- Program states are displayed in the program list. Stopped programs are marked in red. Suspended programs are marked in blue.
- Values of variables are visible in the variable editor. Double click on a variable name to force or lock the variable.
- Values of variables, contacts and coils are displayed in FBD diagrams. Double click on a variable name to force or lock the variable.
- Values of variables, contacts and coils are displayed in LD diagrams. Double click on a variable name to force or lock the variable.
- Step activities (tokens) are displayed in the SFC editor.

In the text (ST or IL) editor, place the mouse cursor on a variable name to display its real time value in a tooltip. Double click on the variable name with the SHIFT key pressed to force or lock the variable.

1.12.2.1. Locking variables

When you double click on a variable symbol during debug, a dialog box is open and enables you to change the value of the variable. It also enables you to *Lock* or *Unlocked* variable. When a variable is locked, its value is no more changed by the runtime. You can then force its value from the debugger independently from the runtime operations. Note that the set of variables that accept to be locked depends on the build options.

Use the **Project / Locked variables** menu command to display the list of variables currently locked. This box also enables to unlock all currently locked variables.



1.12.3. The Watch window

In addition to programs, the Workbench enables you to create monitoring files for testing your applications. There are three styles of documents:

- Watch list: a list of variables
- Recipe: a recipe is a list of variables completed with one or more predefined sets of values

Graphic layouts with animated objects

Lists, recipes and graphics are managed and open from the **workspace** area on the left of the Workbench window. Use the **Insert** commands of the right click popup menu to create a document. Then double click on it in the workspace to open it.

1.12.3.1. Managing lists

To insert a variable, use the **Insert Variable** popup menu command. Alternatively, you can drag variables from program editors or from the variable editor to the Watch window to insert them in the list.

Commands of the local toolbar allows you to modify or arrange the contents of the list. Use the **Move** commands to change the order of variables in the list. Lists can be created and edited in both off line and on line modes.

1.12.3.2. Forcing a variable

At run time, double click on the value of the variable in the list or press <Enter> key when it is selected. A small box appears and allows you to force / control the selected variable.

1.12.3.3. Dumping values

At run time, the value of a variable is displayed in text format in the Value column. You can place the mouse cursor on the value to display it in hexadecimal format. This feature is very useful when you spy strings with a large length or including non printable characters.

1.12.3.4. Recipes

Use the File / New / Recipe menu command to create a recipe. In addition to the list of variables and their values, extra columns can be added in the list for defining sets of values. To enter a value in a set, you just need to select the corresponding cell and enter the value. Some values may be missing in a set. Commands of the local toolbar enable you to create, rename or delete columns. The **Send Recipe** button is used for applying all the values of the selected set to the variables of the list.



Attention:

- Use of recipes from the Workbench ensures that all variables are written together at the same moment in the runtime, i.e. in between two cycles. This leads to a limit of around 50 variables (or less if strings) sent together. This limit does not occur when the recipe is handled directly by the program with the `ApplyRecipeColumn` function.
- Sending a recipe to the target requires special protocol. If the target is T5 runtime 1.21 or older, the request will be denied by the target system.



1.12.4. Command line debugging

The workbench includes a powerful tool that enables you to run debug and monitoring commands during simulation or connected mode tests. Commands are entered as in a console in *prompt* mode. Commands are available for reading and writing any complex variable or expression.

Debugging in prompt mode is available from the **Prompt** tab of the output window.

1.12.4.1. Reading expressions

To read a variable or a complex ST expression, just enter it at the prompt and hit <Enter>. The expression may contain any ST operator, and can be used with variable indexes in case of an array.

Example

```
>MyVar
= 100
>iLevel > 123.4
= FALSE
>MyArray[i+1]
= 123
>
```



The following ST operators can be used in expressions:

Operator	Description
& AND OR XOR	Boolean operators
+ - / *	arithmetics
= <> < > <= >=	comparisons
[,]	array index specification
.	structure member specification

The ST expression can also include constant values.

1.12.4.2. Forcing variables

To force a variable, enter its name followed by the ":" sign and any ST expression. The specification of the forced variable may include variable indexes in case of an array.

Example

```
>MyVar := 200
OK
>MyArray[i+1] := iValue + 2
OK
>
```

1.12.4.3. Specifying a parent program

In order to get access to local variables, you need to specify the name of the default parent program. For that, enter "!local=" followed by the name of the program. When a parent program is specified, all variables used in commands can be either global or local to this program.

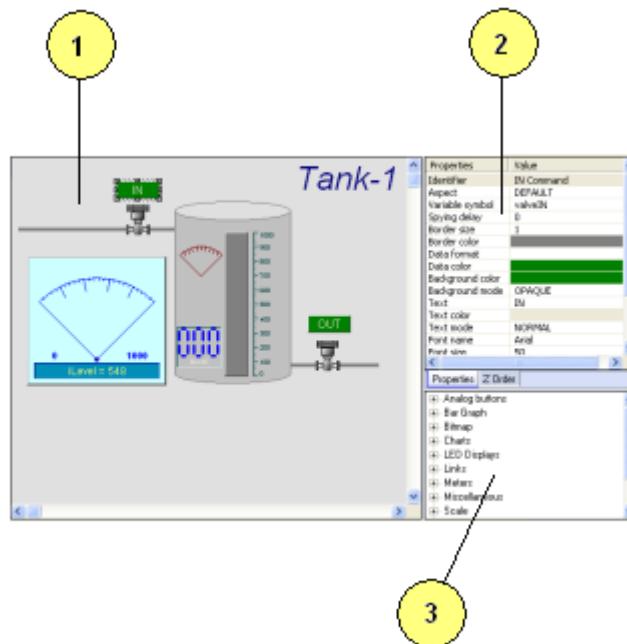
Example

```
>!local=PROG1
>LocalVar
=123
>
```

1.12.5. Using the watch window for graphic monitoring

The Watch window enables you to create graphic pictures animated during simulation or On Line test with real time values of the project variables. To create a new graphic document, Use the **Insert New Graphic** popup menu command in the workspace.

See the list of available objects and their properties for more details on possible drawings.



Graphic area.

Properties of the selected object / arrangement of the Z-order.

Available types of animated objects.

Graphics can also be exported as HTML pages to be displayed in a browser.

To insert a new object, simply drag it from the list of available objects (3) to the graphic area (4). You can then select it to move/resize it, or to enter its properties in the correponding grid (2). Graphic documents are suffixed with ".gra" and can be stored anywhere on the disk.

Below are the commands available from the graphic editor toolbar:

Icon	Description
	Set Operate or Edit mode (see note below).
	Select the previous item in the graphic area.
	Select the next item in the graphic area.
	Align selected items on the left.
	Align selected items on the top.
	Align selected items on the right.



Icon	Description
	Align selected items on the bottom.
	Makes all selected items the same width (*).
	Makes all selected items the same height (*).
	Makes all selected items the same width and height. In those commands, the item used as a reference for sizing is the one selected with grey sizing marks on its border. Click the desired item with CTRL key pressed to select the reference item.
	Send to front: move the selected item to the top in Z order.
	Send to back: move the selected item to the bottom in Z order.
	Define the background color for the graphic area.

The **Operate** button is used to enable/disable changes in the graphic when used On Line. When the operate mode is selected, no change can be made. In that mode, the mouse can be used for driving active objects such as buttons.

The **Z-order** tab in the property area shows the list of the graphic items sorted according to their Z order. You can simply move objects in that list to change the Z-order and thus arrange overlapping items.

1.12.6. Graphic objects

Below are available basic objects you can insert in your graphics:

Basic shapes



A collection of basic drawings is available. Each object may be either static, or linked to a variable used to enable its visibility (show/hide).

Properties

Identifier	Aspect
Variable symbol	Spying delay
Border size	Border color
Data format	Color when not connected
Background color	Background mode
Text	Text color
Text mode	Font name
Font size	TRUE color
FALSE color	Direction

Bitmaps

Bitmap file (BMP, GIF, JPG) can be inserted in the graphic area.

Properties

Identifier	Border size
Border color	Border style
Background color	Background mode
Text	Text color
Text mode	Font name
Font size	Pathname

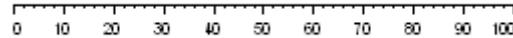
Bitmap display mode



Attention

Large bitmaps are time consuming during animation and can lead to poor performances, mainly if they have the **STRETCH** display mode or the **TRANS** (transparent) background mode.

Scales



Scales are static drawings representing a X or Y axis, generally used to document other objects such as trend charts or bargraphs.



Properties

Identifier	Border size
Border color	Border style
Background color	Background mode
Text	Text color
Text mode	Font name
Font size	Minimum value
Maximum value	Direction
Placement	Nb divisions (main)
Nb divisions (small)	Scale color

Text boxes

Edit **Hello**

Static, animated or edit text boxes are available for displaying / forcing variables.
For edit boxes at runtime, double click on the object to enter the value and then hit <Enter> to validate the input.

Properties

Identifier	Variable symbol
Spying delay	Border size
Border color	Border style
Data format	Background color
Background mode	Text
Text color	Text mode
Font name	Font size
Action	

Switches and 2-state displays



Buttons, switches and 2-state displays are used for control or display of a Boolean variable.

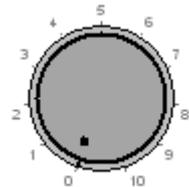
Properties

Identifier	Aspect
Variable symbol	Spying delay
Border size	Border color
Data format	Data color
Background color	Background mode



Text	Text color
Text mode	Font name
Font size	Action
Pathname for <i>TRUE</i> state	Pathname for <i>FALSE</i> state
Bitmap display mode	

Analog buttons



Analog buttons are used for setting the value of an integer or real variable. Mouse is used for setting the value.

Properties

Identifier	Variable symbol
Spying delay	Border size
Border color	Border style
Data format	Data color
Background color	Background mode
Text	Text color
Text mode	Font name
Font size	Minimum value
Maximum value	Scale color

Bar-graphs



Bar graphs are rectangles filled according to the value of an analog variable. Bar graphs can be horizontal or vertical.

Properties

Identifier	Variable symbol
Spying delay	Border size
Border color	Border style
Data format	Data color
Background color	Background mode
Text	Text color

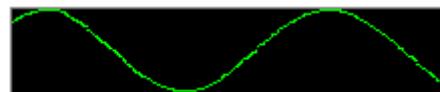


Text mode Font name

Font size Minimum value

Maximum value Direction

Trend charts



Trend charts enable the tracing of a variable as with an oscilloscope.

Properties

Identifier Aspect

Variable symbol Spying delay

Border size Border color

Border style Data format

Data color Background color

Background mode Text

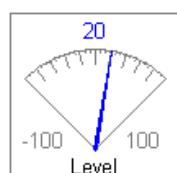
Text color Text mode

Font name Font size

Minimum value Maximum value

Nb of points

Analog meters



Analog meters provide a graphical display of an analog value.

Properties

Identifier Variable symbol

Spying delay Border size

Border color Border style

Data format Data color

Background color Background mode

Text Text color

Text mode Font name

Font size Minimum value



Maximum value	Scale color
Nb divisions (main)	Nb divisions (small)

Sliders



Sliders are used for entering an analog value with a horirontal or vertical mouse driven cursor.

Properties

Identifier	Variable symbol
Spying delay	Border size
Border color	Border style
Data format	Data color
Background color	Background mode
Text	Text color
Text mode	Font name
Font size	Minimum value
Maximum value	Scale color
Direction	

Digital meters



Digital meters (digits) display the value of a variable with the same aspect as a digital clock.

Properties

Identifier	Aspect
Variable symbol	Spying delay
Border size	Border color
Border style	Data format
Data color	Background color
Background mode	Text
Text color	Text mode
Font name	Font size
Minimum value	Maximum value

Links

[Back to main page](#)



Links are mouse driven hyperlinks that are used as shortcuts for opening another graphic document. Using links enable the design of multi page animated applications.

Properties

Border size	Identifier
Background color	Border color
Text	Background mode
Text mode	Text color
Font size	Font name
Link	

Connection status Connection status is a box actuated with the current status of the connection and the connected runtime application. It is aimed for diagnostic.

Properties

Identifier	Spying delay
Border size	Border color
Border style	Data format
Data color	Background color
Background mode	Text
Text color	Text mode
Font name	Font size



1.12.7. Graphic objects properties

This page details all possible properties for graphic objects. Refer to the list of available objects for further information about which property is used for which object.

Identifier	You can freely attach a text identifier to each graphic object inserted in a document. Identifiers are useful for arranging overlapped objects as they appear in the Z-order list.
Variable symbol	This is the full name of the application variable connected to the graphic object. In case of a local variable, its symbol must be prefixed with the parent program name, separated with "/". Example: "MyProg/MyVar".
Spying delay	This is the minimum period for actuating the value of the connected variable, expressed as a number of milliseconds. If the delay is not specified or equal to 0, refresh is done as fast as possible.
Border size	This property indicates the width of the border drawn around the object, as a number of pixels. If this property is 0, then no border is drawn.
Border color	This property indicates the color of the border drawn around the object.
Border style	This property indicates the possible 3D effect used for drawing the border around the object. Possible values are: FLAT = no 3D effect 3DUP = depressed 3D effect 3DDOWN = pressed 3D effect 3D = default 3D effect
Text color	This property indicates the color used for drawing texts in the graphic object.

1.12.7.1. Text mode

This property indicates the font effect used for drawing texts in the graphic object. Possible values are:

HIDE = text is not displayed

NORMAL = normal font

BOLD = bold text

ITALIC = italic text

UNDERLINE = underlined text

Font name	This property indicates the name of the character font used for drawing texts in the graphic object.
Font size	This property indicates the size of the character font used for drawing texts in the graphic object. The size is expressed as a percentage of the actual height of the object. Maximum possible value is 100. This ensures that the ratio is kept when the object is resized.



1.12.7.2. Background color

This property indicates the color used for filling the background of the object. In case of a bitmap, it specifies the color that should not be drawn if the *TRANS* (transparent) background mode is specified.

The Background Mode indicates whether the background of the object must be filled or not. If this property is OPAQUE, then the background is filled with the specified background color. If this property is *TRANS* (transparent) then background is not filled. Transparent drawing mode may be useful in case of overlapping objects.



Attention

Specifying the *TRANS* (transparent) mode for large bitmaps is time consuming and will affect the real time performances of graphic updates.

1.12.7.3. Data format

If defined, this property indicates that the value of the connected variable must be displayed on the graphic object. You must specify for this property a format string that indicates how the data must be formatted.



Attention

The text property is ignored when a data format is specified.

Format string has the same format as the famous "printf" function of "C" language. It may include static characters together with one of the following possible pragmas that specify the value:

%s = default formatting according to IEC syntax

%d = integer (decimal)

%X = hexadecimal

%g = floating point

%.nf = decimal real (*n* is the number of displayed decimal digits)



Example

Format	Value	Displayed String
<i>&d</i>	12.3	12
Var = <i>%g</i> meters	1.2	Var = 1.2 meters
<i>%.2f</i>	1.12345	1.12



Info:

Only one % pragma can be used in a string.

Text

If defined, this property indicates the text to be displayed on the graphic object.



Attention This property is ignored when a data format is specified.

Bitmap display mode

For bitmap based objects, this property indicates whether the attached bitmap



must keep its original aspect or be stretched to the actual size of the object.
Possible values are:

ORIGINAL = Keep the original aspect of the bitmap (cut if too large).
STRETCH = Stretch or shrink the bitmap for fitting the actual size of the graphic object.

 **Attention**

Large bitmaps with *STRETCH* display mode are time consuming during animation and can lead to poor performances.

Minimum value	For analog animated objects (meters, bargraphs, trends...) this property indicates the minimum possible value that can be displayed. For static scales, it indicates the value of the lowest mark.
Maximum value	For analog animated objects (meters, bargraphs, trends...) this property indicates the maximum possible value that can be displayed. For static scales, it indicates the value of the highest mark.
Data color	This property indicates the color used to represent the value of connected variable within the object (for Example the filled part of a bargraph).
Nb divisions (main)	For objects including a graphic scale, this property indicates the number of main division marks to be drawn in the scale.
Nb divisions (small)	For objects including a graphic scale, this property indicates the number of small division marks to be drawn in the scale, between each main division mark.
Scale color	For objects including a graphic scale, this property indicates the color used for drawing the axis, the division marks and corresponding values of the scale.
Bitmap pathname	For bitmaps, this property specifies the pathname of the bitmap to be displayed. BMP, GIF and JPG formats are supported. If no directory is specified, the specified file name is searched: <ul style="list-style-type: none">• In the project folder• In the "\BITMAP" folder of the workbench
Bitmap for "TRUE" state	For 2-state objects having the <i>CUSTOM</i> aspect, this property specifies the pathname of the bitmap to be displayed when the value of the attached variable is <i>TRUE</i> (or not zero for analogs). BMP, GIF and JPG formats are supported. If no directory is specified, the specified file name is searched: <ul style="list-style-type: none">• In the project folder• In the "\BITMAP" folder of the workbench
Bitmap for "FALSE" state	For 2-state objects having the <i>CUSTOM</i> aspect, this property specifies the pathname of the bitmap to be displayed when the value of the attached variable is <i>FALSE</i> (or zero for analogs). BMP, GIF and JPG formats are



	<p>supported. If no directory is specified, the specified file name is searched:</p> <ul style="list-style-type: none">• In the project folder• In the "\BITMAP" folder of the workbench
Color when not connected	For shapes, this property indicates the color used for filling shapes when no variable is attached to the graphic object.
TRUE color	For shapes, this property indicates the color used for filling shapes when the attached variable has the <i>TRUE</i> state, or non zero for analogs.
FALSE color	For shapes, this property indicates the color used for filling shapes when the attached variable has the <i>FALSE</i> state, or zero for analogs.
Direction (basic shapes)	For oriented shapes such as triangles, half ellipses or cylinder, this property indicates the direction of the drawing; to the left, to the right, to the top or to the bottom.
Direction (scale)	For scales, this property indicates the direction of the axis. If <i>LEFT</i> , the minimum value is on the left side. If <i>RIGHT</i> , the minimum value is on the right side.
Placement (scale)	For scales, this property indicates the location of the scale within the object rectangle: on the left, on the right, on the top or at the bottom.
Action (text)	Indicates the possible mouse action for text boxes. Following values are possible: <i>STATIC</i> = No mouse action. <i>EDIT</i> = Double click opens an edit box for entering the variable value.
Action (switch)	Indicates the possible mouse action for switches. Following values are possible: <i>STATIC</i> = No mouse action. <i>PUSHBUTTON</i> = The variable is forced to <i>TRUE</i> when pressed and to <i>FALSE</i> when depressed. <i>SWITCH</i> = The status of the variable is inverted when the button is pressed. <i>ONESHOTBUTTON</i> = Same as switch, but the display remains depressed when the mouse is released.
Direction (bargraph)	For bargraphs, this property indicates the growing direction: to the left, to the right, to the top or to the bottom.
Nb of points (trends)	For trend charts, this property indicates the maximum number of stored points. If the width of the object (in pixels) is less than this number, then oldest points are not visible.
Direction (slider)	For slider, this property indicates whether the slider is horizontal (<i>RIGHT</i>) or vertical (<i>TOP</i>).
Link	This property indicates the name of the target .GRA animated document for shortcuts. If no directory is specified in the link, then the file is searched in the



project folder.

Aspect (shapes)

This property indicates the type of basic shape to be drawn. Possible aspects are:

<i>CYLINDER</i>	= A 3D like cylinder.
<i>ELLIPSE</i>	= An ellipse.
<i>HALFELLPISE</i>	= One half of an ellipse.
<i>GATE</i>	= A simple vector drawing for a valve.
<i>RECTANGLE</i>	= A rectangle.
<i>ROUNDRECT</i>	= A rectangle with rounded corners.
<i>TRIANGLE</i>	= A triangle.

Aspect (switches)

This property indicates the type of switch to be drawn. Possible aspects are:

- DEFAULT* = A standard Windows-like push button.
CUSTOM = A button with *TRUE* and *FALSE* drawings defined with bitmaps.

Aspect (trend charts)

This property indicates the type of drawing for a trend chart. Possible aspects are:

<i>POINT</i>	= Only relevant dots are drawn.
<i>LINE</i>	= Lines are drawn from point to point.
<i>HISTO</i>	= Histogram style.

Aspect (digits)

This property indicates the type of drawing for a digital meter. Possible aspects are:

<i>DEFAULT</i>	= Plain drawing.
<i>BEZEL</i>	= All segments have a 3D effect.



1.12.8. Export graphics as HTML pages

When an animated graphic picture is designed in the workbench, you can export it as a HTML page to be displayed in a browser. Displaying the graphics will require ActiveX Components to be installed on the PC. Note that these components are automatically installed together with the Workbench.

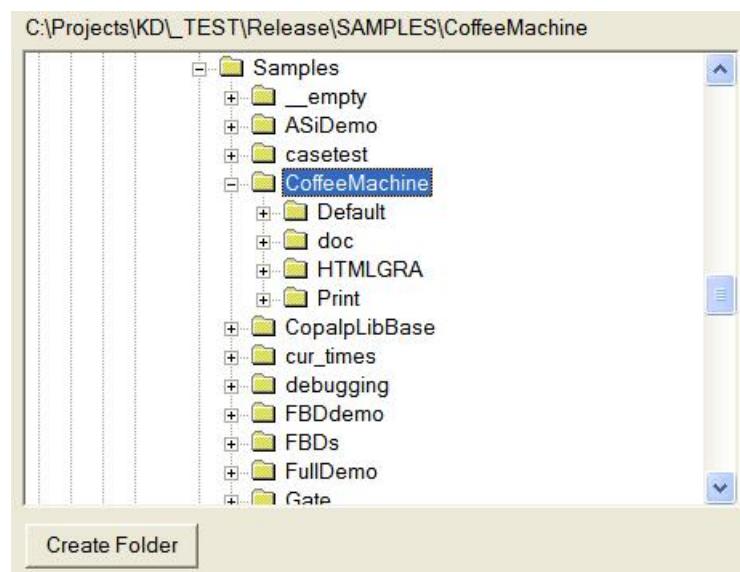
To export a graphic as HTML:

1. Turn the Workbench *Off Line*.

Open the graphic document.

Run the menu command: Tools / Generalte HTML Graphic.

A wizard guides you through the steps for the creation of the HTML page. Press the **Next** button in the welcome page. The following page is displayed:



In this page you must select or create the destination folder where the HTML page and related files will be stored. It is strongly recommended to store the page in an empty folder. Select the target directory and press **Next**. The following page is displayed:

The HTML graphic is now ready to be generated.
Press Next to start process.

Graphic file:	C:\Projects\KD_TEST\Releas... Machine.gra
HTML Path:	C:\Projects\KD_TEST\Rele... CoffeeMachine
HTML file name:	FullDemo.htm
Target name:	K5NET5.DLL
Connection settings:	127.0.0.1
Width:	800
Height:	600
ZIP file name:	[empty]
Include symbol table	<input checked="" type="checkbox"/>
Scroll bar	<input checked="" type="checkbox"/>

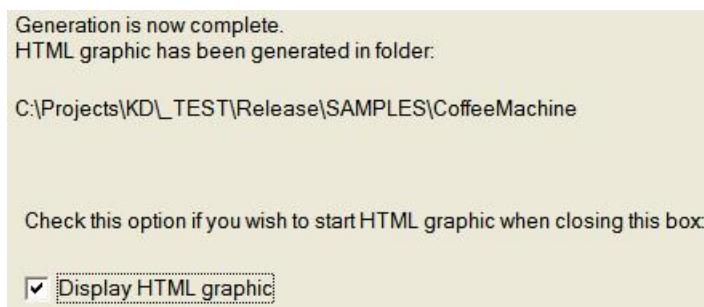


You must fill the following options with relevant values:

Option	Description
HTML file name	Name of the generated HTML file.
Target name	Driver used for communication with the runtime. Specify K5NET5.DLL for a standard T5 runtime.
Connection settings	Communication parameters for connecting to the runtime. In case of an IP address, must be followed by ":" and the IP port number. If not specified, the default port number used is 1100.
Width / Height	Size in pixel of the graphic area in the HTML page.
ZIP file name	Embedded document file name (see notes) This field must be empty if you do not want to download the page to the runtime system .
Include symbol table	If this option is checked, the symbol table used for communication is saved together with the HTML page. If not, symbols are uploaded when the HTML page is opened.
Scroll bar	Indicates whether you want the graphic area to include a scroll bar.

If you specify a "ZIP" file name, then all files used by the page are zipped and sent together with the HTML page to the runtime system. This assumes that the runtime system is connected and includes a WEB server. Refer to the instructions of your OEM to know if any path or file name is imposed for the ZIP file. Note that this ZIP file may be big as it may include bitmap pictures.

After you press the **Next** button, the page is built and the following page is displayed:



Check the **Display HTML graphic** option if you want to test right now your page in a browser.

If you have selected a "ZIP" file name for embedding the page to the runtime system, press **Finish** to download the page and attached documents to the runtime. When download is complete, Press **Cancel** to close the box.



1.12.9. Digital sampling trace

The runtime system as the simulator include a digital sampling trace recorder. The recorder is used to register periodically the state of up to 8 Boolean variables. Samples can be registered either on each cycle or according to a configurable period. The digital sampling trace is a useful tool for tracking aleas and events in the runtime application.

The sampling trace can be configured and watched from the Output window. The sampling trace is available only during simulation or on line debugging.



Attention

- The digital sampling trace is a unique resource of the runtime system. The settings of the recorder are the same for all recorded variables.
- The recording is limited to 900 samples of up to 8 BOOL variables.
- Sampling trace recorder may be not available on some targets. Please refer to OEM instructions for further details about available features.
- The recording of the sampling trace is time consuming and may slow down performances of the runtime system.

1.12.9.1. Operations

Use the following commands in the Log window when the **Digital Sampling Trace** tab is focused:

Icon / Shortcut	Description
-----------------	-------------

	Define the variables and the settings of the sampling trace.
	Start recording.
	Stops recording.
	Set or reset the auto-scroll mode.
+ / -	Zoom in or out (num pad).

1.12.9.2. Settings

Before starting a recording, you need to setup the parameters for the recorder. This includes the list of spied variables, a period (either a time or on each cycle), plus start and stop conditions. All variables must have the BOOL data type.

Note that, if you program a period for the recorder, it means "wait at least this time between two samples", and does not correspond to an accurate time synchronization.

Press the **Set** button to validate the settings. Press the **Reset** button to clear the current settings in the runtime system.



1.12.9.3. Start condition

The **Start Condition** tab of the settings box enables you to define which condition will start the recording. The following choices are available:

- Immediately.
- Later: you will have to manually start the recorder using the **Start** command.
- On the rising or falling edge of a BOOL variable, possibly with a delay.

The delay is expressed as a number of samples omitted after the start condition occurred, before the recording actually starts.

1.12.9.4. Stop condition

The **Stop Condition** tab of the settings box enables you to define which condition will stop the recording. The following choices are available:

- Never: you will have to manually stop the recorder using the **Stop** command.
- When the buffer is full.
- On the rising or falling edge of a BOOL variable, possibly with a delay.

The delay is expressed as a number of samples passed after the stop condition, before the recording actually stops.

1.12.9.5. Remarks

The recorder cannot be restarted after points have been registered, even if stopped. To restart the recording, you first have to re-validate the settings.

The sampling trace must be configured or started when the Workbench is used either for simulation or on line debugging.

Use the File / Save As and Edit / Copy commands for exchanging recorded data with other applications such as spreadsheets.



1.12.10. Step by step debugging

In addition to the cycle by cycle execution mode, the debugger has a rich collection of powerful features for making step by step debugging in the source code of your application. The step by step feature is completed by the possibility to place breakpoints in the source code of the application. The debugger also shows you the call stack of called UDFBs and sub-programs when stepping. Step by step debugging is available:

- In ST and IL text programs (a step is a statement).
- In LD program (a step is a rung).
- In FBD (a step is a graphic symbol corresponding to an action).

Step by step debugging is not possible in SFC programs.

Attention

- Step by step debugging is available only if the project has been compiled with the *DEBUG* option. This option can be selected from the project compiling options dialog box.
- An application compiled in *DEBUG* mode includes additional information for stepping. This leads to bigger code size and less performances. It is recommended to compile your application in *RELEASE* mode when the debugging is finished.
- Step by step debugging may be not supported on some target systems. Please refer to OEM instructions for further details about available features.

There are two possibilities for entering the step by step debugging mode:

- Place a breakpoint in a program (using the Build / Set-Remove Breakpoint command). When the breakpoint is reached, the execution stops at the specified location and you can step further in the program.
- When the target is in cycle stepping mode (STOP), you can step at the beginning of the first program.

The following commands are available for stepping, either from the main control panel, or from the **Build** menu of editors:

Command Icon Description



Step over: If the next instruction is a call of a function block or a sub-program, the execution continues up to the end of the called block.



Step in: If the next instruction is a call of a function block or a sub-program, the next step will be at the beginning of the called block.



Step out: If the current stepping position is in a called function block or a sub-program, the execution continues up to the end of the block.

In addition to these commands, you can at any time:

- Execute the cycle (from the current position up to the end of the last program).



- Restart the target in *normal* execution mode (RUN).

1.12.10.1. Breakpoints

To put or remove a breakpoint in a program, use the Build / Set-Remove Breakpoint menu command. If the current position is not on a valid line for stepping, the breakpoint is automatically moved to the nearest valid position.

Breakpoints can be placed in programs, sub-programs or UDFBs. They are not available in SFC programs.

You can see the list of active breakpoints in the Log Window or in the output window of an editor. From here you have the possibility to remove all breakpoints in a single command. At any moment you can double click on a breakpoint in the Log window to open the corresponding program at the breakpoint location.

1.12.10.2. The call stack

When stepping, the Call Stack is available in the Log Window or in the output window of editors. It shows the stack of called function blocks and/or sub-programs, from the top level calling program up to the current stepping position. For a UDFB, the Call Stack also indicates which instance of the UDFB is executed.

At any moment you can double click on a line of the Call Stack to open the corresponding program.

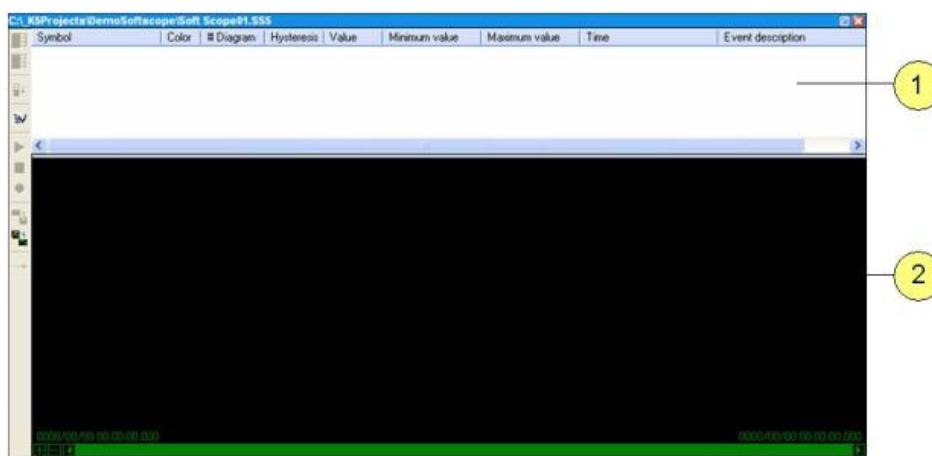


1.12.11. Soft oscilloscope

The soft oscilloscope enables you to track the value of Boolean or numerical variables and display it in a curve. Spied variables are tracked by the runtime, which detects changes and assigns time stamps so that the trend is displayed accurately.

The soft oscilloscope is available during On-Line debugging. It uses the T5 spontaneous protocol (used for binding) over Ethernet. Thererfore, the soft oscilloscope cannot be used with runtime systems having no Ethernet connection.

The soft oscilloscope is run from the Workspace window. Use commands of the contextual popup menu to create new soft scope configurations. Below is the soft oscilloscope screen:



1. List of variables to be displayed.
2. Diagram area (oscilloscope display).

In the diagram area, the user can zoom, explore a particular time range and automatically scroll the diagrams.

The scroll bar behind, represents the current watching time range. The complete width represents the full time since target has started to the current target time.

To add new symbols to be displayed in diagrams, drag and drop them from the variable editor, or double click in an empty line in the list area. These new symbols can be added in both *On Line* or *Off Line* modes.

The following pieces of Information are configured in the list:

Parameter	Description
Symbol	Name of the spied variable.
Color	Color used to draw the curve.
#Diagram	Index of the diagram pane - default is 1. You can define up to 30 panes.
Hysteresis	Hysteresis to apply for change detection of analog values. The hysteresis is entered as an absolute value.
Value	The current value of the variable is refreshed in this column.



Parameter	Description
Minimum/Maximum	Range of the Y axis.
Time	Time and date of the last change.
Description	Free description text.

The following commands are available from the soft oscilloscope toolbar:

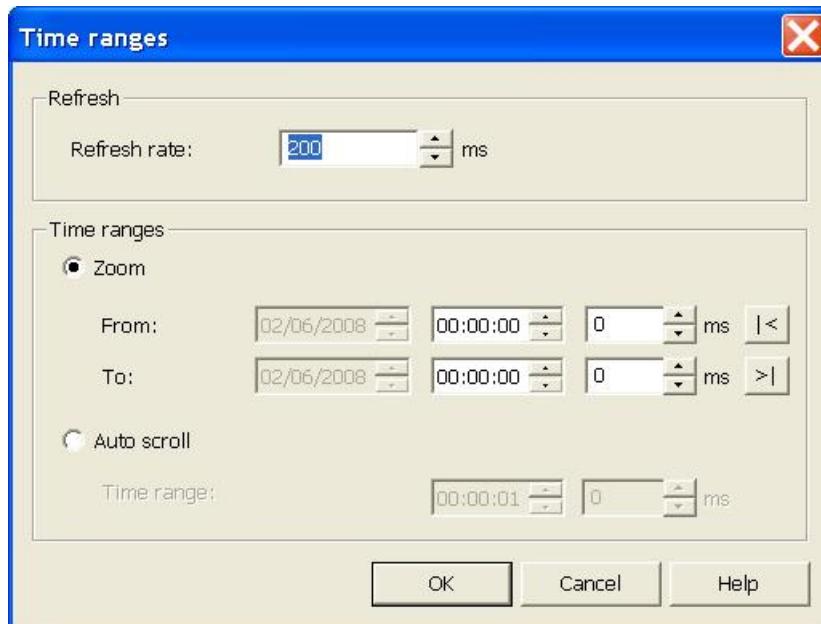
Icon	Description
	Move the selected variable up or down in the list.
	Sort variables of the list.
	Setup time ranges.
	Start the oscilloscope.
	Stop the oscilloscope.
	Start recording.
	Save record to the file.
	Reload record from file.
	Auto-scroll mode (toggle).

When sampling is active, you can start recording all events from now by clicking on corresponding (red) button. You must specify a csv file where samples will be recorded. All events on all symbols will be recorded in this file until you uncheck recording by clicking the button again.

When sampling is inactive, you can save particular parts of the diagrams to the disk. For this, will select the symbol(s) in the list and then zoom the time area to fit the range you want to save. Selected events will be saved in a .rec file. Later, you can restore a .rec file to the diagram. When user restore his diagrams, only symbols existing in the list will be updated.



The **Setup** button opens the following box for configuring the X-axis of the display:



Two modes are available:

- Zoom mode
- Autoscroll mode

You can choose a specific time area to be displayed with from and to lines. Buttons near are used to zoom from the first start time and to the current time.

In autoscroll mode, diagrams are scrolled automatically at the refresh rate to display the specified time range. The refresh rate can be increased for better refresh in autoscroll mode (for instance if the time range is long and if there lots of events occur during this period).



1.13. Elmo Fieldbus configuration

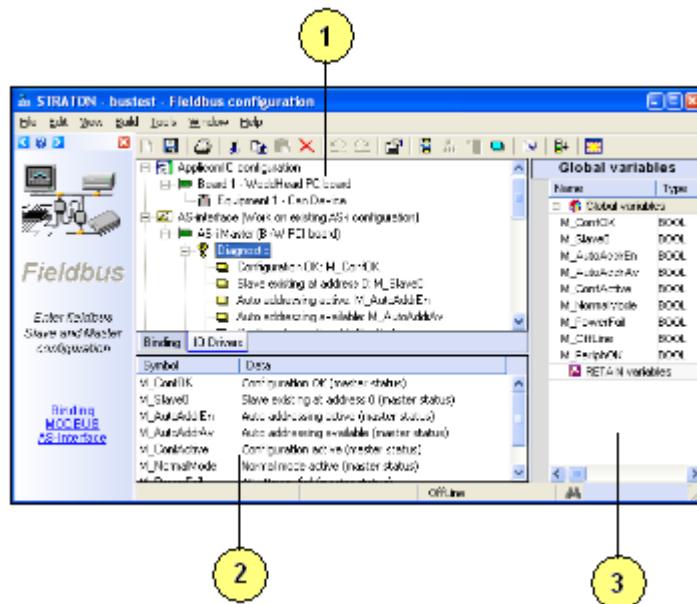
The workbench includes an integrated fieldbus configuration tool for various kinds of networked I/Os and protocols. The configuration tool enables you to describe networks as configuration trees and to wire variables to the I/O channels of devices.

Icon Description



Use the command File / Open / Fieldbus configuration.

The fieldbus configuration proposes the following workspace:



1. Configuration tree.

Items within the node currently selected in the tree.

Variables of the project.

Each kind of fieldbus is shown as a top-level node in the configuration tree. Run the Edit / Insert Configuration menu command to select a configuration to be added to the tree. Each configuration is displayed as a 4 level tree having the following form:

- Configuration (kind of fieldbus)
- Communication port or master device
- Slave device or data block
- Connected variable



Use the following buttons in the toolbar for building the configuration tree:

Icon Description



Insert a new fieldbus (top level) in the configuration.



Insert a new master/port node in the selected fieldbus.



Insert a new slave/data block node under the selected master.



Insert a new variable node under the selected slave.

When an item is selected in the tree, all its children can be edited in the grid below. Alternatively you can double click on an item in the tree to enter its properties in a dialog box. Use the View / Grid menu command to show or hide the grid area.

You can also drag a variable from the list of declared variables (on the right) directly to a slave item in the tree.



1.14. Tools

In addition to programming and test features, the workbench offers a collection of tab tools for managing and maintaining projects:

- Import / export projects
- Comparing projects
- Building HTML documents
- Monitoring applications
- The Console



1.14.1. Import / Export projects

The Workbench enables archiving of projects for exchange purposes. An archive is a unique compressed ZIP file containing all the files of a project.

Use the commands of the **File** menu in the main window to save the project to a ZIP file or open a project from a ZIP file.

When you export a project, you can select to embed in the archive the definition of the OEM library elements (I/O devices, "C" functions and function blocks) referenced in the project. It is recommended to check this option if you are using custom functions, blocks or I/O devices that may not be installed on other machines.

When you import a ZIP archive, the workbench builds a new project where the archive is decompressed. You can select the name of the created project and its location on the disk. If you decide to import to your local library the OEM library items embedded in the archive, you will have to confirm the copy in case of an overwrite. You can select in which OEM library you want to put imported library elements.

Note that it is not possible to overwrite in the import library an item currently defined in another library.

1.14.2. Monitoring applications

The workbench includes a wizard that build a monitoring application for the open project. The monitoring application is operated by the Monitoring Tool for spying and controlling the project without need of a full Workbench. Run the Tools / Build monitoring application from the menu of the main window to start the wizard.

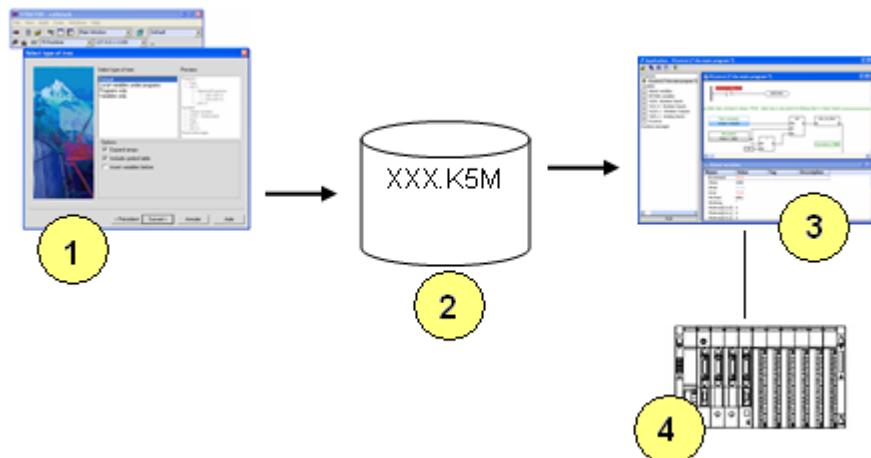


Figure 1.2: Principles of monitoring applications

The Workbench (1) is used for building the monitoring application in a unique file (2). The standalone Monitoring Tool (3) operates the monitoring application: it animates documents according to real time values from the connected target runtime (4). A monitoring application groups programs, lists of variables, plus the trace of runtime messages. The tool displays the value of the variables and can be used for forcing a variable. The set of items available in the monitoring application is selected in the wizard when building the application.

Below is the description of the following operations you must perform when using the wizard.

To monitor an application:

1. Set a name to the application. This name will be displayed in the main title bar of the viewer. The configuration of a monitoring application can be saved on the Workbench project. You can select to reopen an existing configuration in this page.
2. Set the main options for the application. The list of visible items will be shown as a tree in the viewer. You can choose among various presentations for the tree.
If you check the **Include symbol table** option, the monitoring application will work on the full symbol table generated by the compiler. You must take care that the same version of the application is running on the target. If you don't select this option, then the monitoring tool will automatically upload symbols from the target at connection time. In that case, you must ensure that all used symbols are embedded in the runtime application.
3. Select all the items of the project to be included in the monitoring application.
4. For each item included in the application a password can be defined, so that the corresponding document will be protected in the monitoring application.
5. You can define passwords for the Monitoring Tool to make write access (forcing) to runtime variables.

For each variable you can select one of the following access protection:



- Free : the variable can be freely forced.
- Protected : forcing the variable is possible with a password.
- No : the variable can never be forced.

Simply drag variables from the list at the bottom to the upper list to set its protection mode. The *Default* choice indicates the protection mode that should be applied to all variables not explicitly expressed in the upper list.

6. Everything is now ready to generate the monitoring application. Select the pathname of the file that will contain the monitoring application. The application is stored as a unique compressed file.
7. The application is now generated. You can run it in simulation mode in the last page of the wizard. This page also proposes you to save the current configuration of the application so that you can use it later for update purpose.

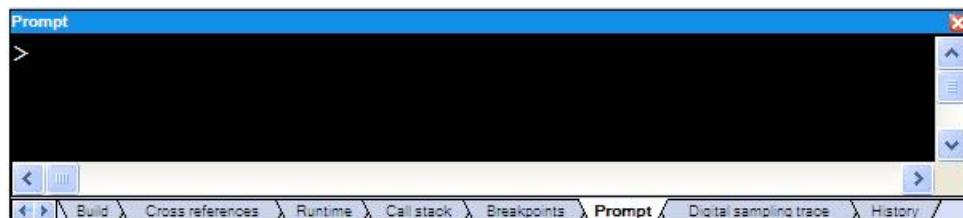
Note: All programs stored in the monitoring application are encrypted. The user of the viewer cannot explore, copy or drag the contents of a watched program.



1.14.3. The console

The Workbench includes a powerful *console* where you can, during the development of a project, run some operations in *command line* text mode instead of using menus. This may be very useful to automate some repetitive tasks or quickly manipulate objects of huge projects.

While editing, go in the **Prompt** tab of the output window:



Enter the command "?" to get the list of available commands and their syntax. Below are available commands, sorted by categories.

1.14.3.1. Special commands

? / HELP

Display help

Syntax:

HELP [command] ? [command]

Arguments:

command : name of the command you want help about



Info

If no "command" argument is specified, then HELP lists all possible commands.

FOR

Iterate a command

Syntax:

Arguments:

FOR <min> TO <max> <command> [args]

min : first enumeration integer value (see notes)

max : last enumeration integer value (see notes)

command : command to execute

args : arguments of the command



Info

In command arguments, you can use the special '%' characters, to be replaced with the iteration number. If you specify consecutive '%' characters, this represents the iteration index formatted on several characters, completed on the left by '0' digits'. For Example, the command:

FOR 1 TO 3 CreateProgram LD Prog%%

creates 3 LD programs called Prog01, Prog02 and Prog03

Note that not all commands support FOR iterations. See details about specific



commands for further information.

1.14.3.2. Managing Programs

CreateProgram

Create a main program

Syntax

CreateProgram <language> <name>

Arguments

language : SFC or FBD or LD or ST or IL

name : program name



Info

The program is created at the end (after the last existing program) of the cycle. This command can be used in a "FOR" loop.

CreateSP

Create a sub-program

Syntax

CreateSP <language> <name>

Arguments

language : FBD or LD or ST or IL

name : sub-program name



Info

This command can be used in a "FOR" loop.

CreateUDFB

Create a User Defined Function Block

Syntax

CreateUDFB <language> <name>

Arguments

language : FBD or LD or ST or IL

name : function block name



Info

This command can be used in a "FOR" loop.

CreateSfcChild

Create a child SFC program

Syntax

CreateSfcChild <parent> <name>

Arguments

parent : name of the parent SFC program

name : sub-program name



Info

This command can be used in a "FOR" loop.

CopyProgram

Duplicate a program

Syntax

CopyProgram <program> <newprogram>

Arguments

program : name of the source program

newprogram : name of the destination program to be created



Info



This command is used for creating copies of a program. It should not be used for overwriting an existing program.

This command can be used in a "FOR" loop.

DeleteProgram

Delete a program

Syntax

DeleteProgram <program>

Arguments

program : program to delete - may contains '?' wildchars



Info

This command can be used in a "FOR" loop.

EnumProgram

Enumerate programs

Syntax

EnumProgram [filter]

Arguments

filter : filtering strings - may contain '*' and '?' wildchars



Info

This command cannot be used in a "FOR" loop.

GetProgram

Get Information about a program

Syntax

GetProgram <program>

Arguments

program : program name



Info

This command cannot be used in a FOR loop.



1.14.3.3. Managing Program Folders

CreateFolder

Create a folder of programs

Syntax

CreateFolder <name>

Arguments

name : folder name



Info

The folder is created under the root folder of the workspace. Nested folders are not supported by this command.

This command can be used in a "FOR" loop.

SendToFolder

Send a program to a folder

Syntax

SendToFolder <program> <folder>

Arguments

program : name of a program, sub-program or UDFB

folder : name of an existing folder under the root folder



Info

The destination folder must exist and be under the root folder of the workspace. Nested folders are not supported by this command.

This command can be used in a "FOR" loop.

DeleteFolder

Delete a folder

Syntax

DeleteFolder <folder>

Arguments

folder : folder name



Info

The destination folder must exist and be under the root folder of the workspace. Nested folders are not supported by this command.

This command can be used in a "FOR" loop.

EnumFolder

Enumerate folders

Syntax

EnumFolder [filter]

Arguments

filter : filtering strings - may contain '*' and '?' wildchars



Info

This command cannot be used in a FOR loop.



1.14.3.4. Managing Variables

CreateVar

Create a variable

Syntax

CreateVar <name> <type>

Arguments

name : name of the variable - see notes

type : data type (can be a function block to create an instance)



Info

This function is used for declaring a variable, or for adding an item to a data structure. The name of the variable must be prefixed with its group name and the '.' separator. The variable is considered as GLOBAL if no prefix is specified. Below are some examples:

Argument

Description

VarName A global variable

RETAIN.VarName A RETAIN variable

ProgName.LocVarName A variable local to a program

UDFBName.ParamName A parameter of a UDFB

StructName.Item An item of a data structure

This command can be used in a FOR loop.

CreateInParam

Create an input parameter

Syntax

CreateInParam <POU>.<name> <type>

Arguments

POU : name of the sub-program or UDFB

name : name of the parameter

type : data type (can be a function block to create an instance)



This command can be used in a FOR loop.

CreateOutParam

Create an output parameter

Syntax

CreateOutParam <POU>.<name> <type>

Arguments

POU : name of the sub-program or UDFB

name : name of the parameter

type : data type (can be a function block to create an instance)



This command can be used in a FOR loop.

DimVar

Set variable dimensions

Syntax

DimVar <var> <Dim1> [Dim2 [Dim3]]

**Arguments**

var : name of the variable - see notes

Dim1, Dim2, Dim3 : dimensions (from highest to lowest)

**Info**

This function is used for variables, parameters of sub-programs and UDFBs or for items of data structures. The name of the variable must be prefixed with its group name and the '.' separator. The variable is considered as GLOBAL if no prefix is specified. Below are some examples:

Argument	Description
VarName	A global variable
RETAIN.VarName	A RETAIN variable
ProgName.LocVarName	A variable local to a program
UDFBName.ParamName	A parameter of a UDFB
StructName.Item	An item of a data structure

This command can be used in a FOR loop

DeleteVar**Delete a variable****Syntax**

DeleteVar <var>

Arguments

var : name of the variable - see notes

**Info**

This function is used for variables, parameters of sub-programs and UDFBs or for items of data structures. The name of the variable must be prefixed with its group name and the '.' separator. The variable is considered as GLOBAL if no prefix is specified. Below are some examples:

Argument	Description
VarName	A global variable
RETAIN.VarName	A RETAIN variable
ProgName.LocVarName	A variable local to a program
UDFBName.ParamName	A parameter of a UDFB
StructName.Item	An item of a data structure

This command can be used in a FOR loop.

InitVar**Set variable initial value****Syntax**

InitVar <var> [value]

Arguments

var: name of the variable - see notes



value: initial value (initial value is removed if this argument is omitted)



Info

This function is used for variables, parameters of sub-programs and UDFBs or for items of data structures. The name of the variable must be prefixed with its group name and the '.' separator. The variable is considered as GLOBAL if no prefix is specified. Below are some Examples:

Argument	Description
VarName	A global variable
RETAIN.VarName	A RETAIN variable
ProgName.LocVarName	A variable local to a program
UDFBName.ParamName	A parameter of a UDFB
StructName.Item	An item of a data structure

This command can be used in a FOR loop.

SybVar

Specifies whether the symbol of a variable must be embedded

Syntax

SybVar <var> < ON | OFF >

Arguments

var: name of the variable - may contain '*' and '?' wildchars - see notes



Info

This function is used for variables, parameters of sub-programs and UDFBs or for items of data structures. The name of the variable must be prefixed with its group name and the '.' separator. The variable is considered as GLOBAL if no prefix is specified. Below are some examples:

Argument	Description
VarName	A global variable
RETAIN.VarName	A RETAIN variable
ProgName.LocVarName	A variable local to a program
UDFBName.ParamName	A parameter of a UDFB
StructName.Item	An item of a data structure

This command can be used in a FOR loop.

ProfileVar

Set variable profile and embedded properties

Syntax

ProfileVar <var> [profile [props]]

Arguments

var : name of the variable - may contain '*' and '?' wildchars - see notes

profile : profile name - the profile is removed if not specified



pros : embedded properties - properties are removed if not specified



Info

This function is used for variables, parameters of sub-programs and UDFBs or for items of data structures. The name of the variable must be prefixed with its group name and the '.' separator. The variable is considered as GLOBAL if no prefix is specified. Below are some examples:

Argument	Description
VarName	A global variable
RETAIN.VarName	A RETAIN variable
ProgName.LocVarName	A variable local to a program
UDFBName.ParamName	A parameter of a UDFB
StructName.Item	An item of a data structure

This command can be used in a FOR loop.

EnumVar Enumerate variables

Syntax EnumVar [filter]

Arguments filter : filtering strings - may contain '*' and '?' wildchars



Info

This function is used for variables, parameters of sub-programs and UDFBs or for items of data structures. The filtering string must be prefixed with its group name and the '.' separator. Variables are considered as GLOBAL if no prefix is specified. Below are some examples:

Argument	Description
*	A global variable.
RETAIN.*	A RETAIN variable.
ProgName.*	A variable local to a program.
UDFBName.*	A parameter of a UDFB.
StructName.*	An item of a data structure.

This command cannot be used in a FOR loop.

GetVar

Get Information about a variable

Syntax GetVar <var>

Arguments var : name of the variable - see notes

 Info

This function is used for variables, parameters of sub-programs and UDFBs or for items of data structures. The name of the variable must be prefixed with its group name and the '.' separator. The variable is considered as GLOBAL if no prefix is specified. Below are some examples:

Argument	Description
VarName	A global variable
RETAIN.VarName	A RETAIN variable
ProgName.LocVarName	A variable local to a program
UDFBName.ParamName	A parameter of a UDFB
StructName.Item	An item of a data structure

This command cannot be used in a FOR loop.

GetVarProp**Get variable embedded properties****Syntax**

GetVarProp <var>

Arguments

var : name of the variable - see notes

 Info

This function is used for variables, parameters of sub-programs and UDFBs or for items of data structures. The name of the variable must be prefixed with its group name and the '.' separator. The variable is considered as GLOBAL if no prefix is specified. Below are some examples:

Argument	Description
VarName	A global variable.
RETAIN.VarName	A RETAIN variable.
ProgName.LocVarName	A variable local to a program.
UDFBName.ParamName	A parameter of a UDFB.
StructName.Item	An item of a data structure.

This command cannot be used in a FOR loop.



1.14.3.5. Managing Data Types (structures)

CreateStruct Create a structure

Syntax CreateStruct <name>

Arguments name : structure name



Info This command can be used in a FOR loop.

DeleteStruct Delete a structure

Syntax DeleteStruct <struct>

Arguments struct : structure name



Info This command can be used in a FOR loop.

EnumStruct Enumerate structures

Syntax EnumStruct [filter]

Arguments filter : filtering strings - may contain '*' and '?' wildchars



Info This command cannot be used in a FOR loop.



1.14.3.6. Managing I/O boards

CreateIO Create an instance of an IO device

Syntax CreateIO <slot> <IOType>

Arguments %board : address of the IO board (e.g. %QX0)

name : name of the parameter

value : value for the parameter



Info This command cannot be used in a FOR loop.

IOParam Set a parameter of an IO board

Syntax IOParam <%board> <name> <value>

Arguments struct : structure name



Info This command can be used in a FOR loop.

IOAlias Give an alias to an IO channel

Syntax IOAlias <%channel> <alias>

Arguments %channel : address of the IO channel (e.g. %QX0.1)

alias : readable name to be used as an alias



Info This command cannot be used in a FOR loop.

DeleteIO Delete an instance of an IO device

Syntax DeleteIO <slot>

Arguments slot : slot number (0 .. 255)



Info This command cannot be used in a FOR loop.

EnumIO Enumerate IO boards

Syntax EnumIO

Arguments



Info This command cannot be used in a FOR loop.

GetIO Get Information about an IO board

Syntax GetIO <%board>

Arguments %board : address of the IO board (e.g. %QX0)



Info This command cannot be used in a FOR loop.



1.14.3.7. Managing comment texts

Comm Set comment text

Syntax Comm <X|P|S|V|B> <name> <comment>

Arguments name : object name

comment : comment text



Info

Use the following values for the first argument that specifies the kind of object:

X : project

P : program or sub-program or UDFB

S : structure

V : variable

B : IO board

In case of a variable, the name must be prefixed by the group name followed by the '.' separator. Below are some Examples:

Argument	Description
VarName	A global variable.
RETAIN.VarName	A RETAIN variable.
ProgName.LocVarName	A variable local to a program.
UDFBName.ParamName	A parameter of a UDFB.
StructName.Item	An item of a data structure.

This command can be used in a FOR loop, except for IO boards.

Tag Set short comment text (tag) - variables only

Syntax Tag V <name> <comment>

Arguments name : variable name (see notes)

comment : short comment text (tag)



Info

The name of the variable must be prefixed by the group name followed by the '.' separator. Below are some Examples:

Argument	Description
VarName	A global variable
RETAIN.VarName	A RETAIN variable
ProgName.LocVarName	A variable local to a program
UDFBName.ParamName	A parameter of a UDFB

StructName.Item An item of a data structure

This command can be used in a FOR loop, except or IO boards

GetComm Display a comment text

Syntax GetComm <X|P|S|V|B> <name>

Arguments name : object name



Info

Use the following values for the first argument that specifies the kind of object:

X : project

P : program or sub-program or UDFB

S : structure

V : variable

B : IO board

In case of a

the '.' separa

Argument	Description
----------	-------------

Argument	Description
VarName	A global variable
RETAIN.VarName	A RETAIN variable
ProgName.LocVarName	A variable local to a program
UDFBName.ParamName	A parameter of a UDFB
StructName.Item	An item of a data structure

This command can be used in a FOR loop, except or IO boards.

GetTag Display a short comment text (tag) - variables only.

Syntax GetTag V <name>

Arguments name : variable name (see notes)



Info

The name of the variable must be prefixed by the group name followed by the '!' separator. Below are some examples:

Argument	Description
VarName	A global variable
RETAIN.VarName	A RETAIN variable
ProgName.LocVarName	A variable local to a program
UDFBName.ParamName	A parameter of a UDFB



StructName.Item

An item of a data structure

This command can be used in a FOR loop, except or IO boards



1.15. Resources

In addition to IEC-1131 programming, the workbench enables you to design some configuration data to be embedded together with the application code and used at runtime using related functions or function blocks.

The system supports the following kinds of resources

String tables: statically defined tables of texts to be used at runtime with STRING variables.

Signals: analog signals to be played at runtime for generating a setpoint.

To create and edit resources, select the **Resources** tab in the **Workspace** area on the left side of the Workbench window, and then use the commands of the contextual menu to create, rename or delete resource files.



1.15.1. String table resources

String tables are resources (embedded configuration data) edited with the Workbench. A string table is a list of items identified by a name and referring to one or more character strings.

String tables are typically used for defining static texts to be used in the application. The following functions can be used for getting access to string tables in the programs:

StringTable: selects the active string table.

LoadString: Load a string from the active table.

In addition, each string table may contain several columns of texts for each item, and thus ease the localization of application, simply by defining a column for each language. This way the language can be selected dynamically at runtime, simply by specifying the active language (as a column) in the StringTable function.

The name entered in the string table as an "ID" is automatically declared for the compiler and can directly be passed to the LoadString function without re-declaring it. The name must conform to IEC standard naming rules.

Obviously, you could do the same by declaring an array of STRING variables and enter some initial values for all items in the array. By the way, string tables provide significant advantages compared to arrays:

- The editor provides a comfortable view of multiple columns at editing.
- String tables are loaded in the application code and does not require any further RAM memory unlike declared arrays.
- The string table editor automatically declares for you readable IDs for any string item to be used in programs instead of working with hard-coded index values.



Info

If the text is too long for the STRING variable when used at runtime, it is truncated.

You can use special '\$' sequences in strings in order to specify non printable characters, according to the IEC standard:

Code	Meaning
\$\$	A "\$" character
\$'	A single quote
\$T	A tab stop (ASCII code 9)
\$R	A carriage return character (ASCII code 13)
\$L	A line feed character (ASCII code 10)
\$N	Carriage return plus line feed characters (ASCII codes 13 and 10)
\$P	A page break character (ASCII code 12)
\$xx	Any character (xx is the ASCII code expressed on two hexadecimal digits)

1.15.2. Analog signals resources

Analog signals are resources (embedded configuration data) edited with the Workbench. An analog signal is entered as a list of analog points among a time X-axis.

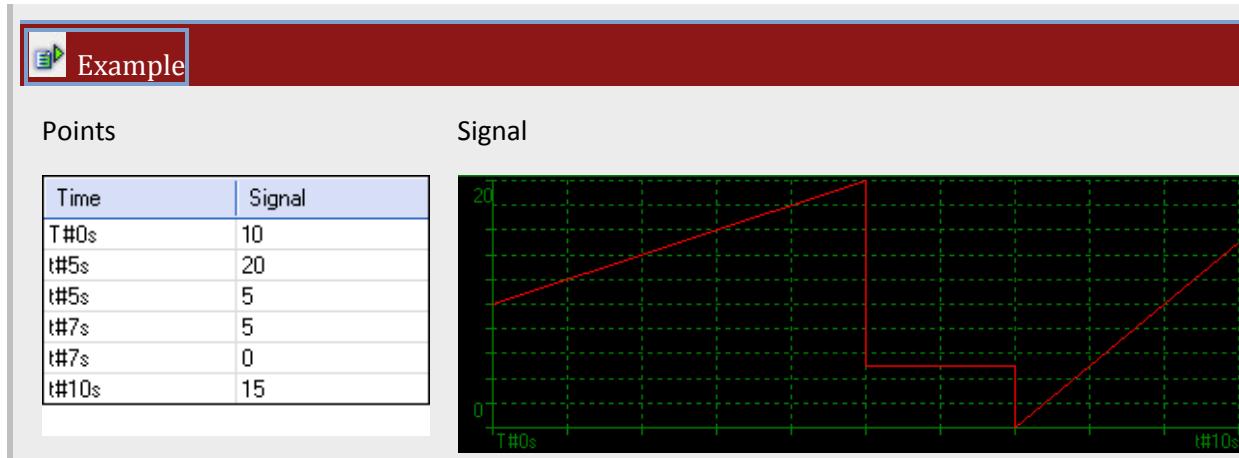
Signals are typically used for statically designing a set-point signal to be played at runtime using the following functions and blocks:

SigID Get the identifier of a signal resource (to be passed to other blocks).

SigPlay Plays a signal.

SigScale Get a point value from a signal.

A signal is entered in the editor as a list of point. Each point refers to a 0-based time value (X axis) and a REAL analog value (Y axis).



In between two consecutive points, the signal must be understood as an analog ramp. If you need to represent a discontiguous change (vertical segment), specify the same time value (X axis) for two consecutive points. The number of points is not limited.



1.16. Uploading projects source code

The workbench enables you to embed on the target the source code of the project, so that it can be uploaded later. Source code is filtered and zipped in order to reduce backup memory requirements. As sending source to the target may involve a significant download time, it is up to you to explicitly activate the download command. Those commands are available from the **File** menu of the main window:

Save Project to Target: zip project source files and send them to the target.

Open project from Target: upload zipped source file from the target and rebuild the project.



1.16.1. Save to target (download)

When saving the project to the target you have to specify the address and communication parameters of the remote runtime system. The following options enable you to send more or less optional Information with project source code:

Symbol table	The symbol table will be required after upload for monitoring variables. If you do not embed the symbol table, you will have to recompile the uploaded application for reading or writing variables.
Debug Information	This file will be required after upload for step by step debugging and use of breakpoints. If you do not embed the symbol table, you will have to recompile the uploaded application for stepping the application.
Spy lists	These are all files created with the Watch Window, such as lists and recipes.
Wizard settings	These are current settings of wizards such as the Monitoring Application Builder.
Project history	This is the list of modifications entered in the project.
Comment texts	All comments entered for variables, programs. Comments within the programs are always saved.
Bitmaps and icons	These are all BMP, GIF, JPG or ICO files stored in the project folder and possibly used in monitoring views.
Referenced OEM library elements	Definition of all the library elements ("C" functions and blocks, I/Os, profiles) actually used in the project.

In addition to standard files, you can specify some *extra* files to be downloaded. In that case, all of them will be located in the loaded project folder after upload, even if they are originally located in other folders.

Removing some options enables you to reduce the size of embedded source code.



1.16.2. Open from target (upload)

When uploading the application, you need to specify the address and communication parameters of the remote runtime system, plus a name and a location for the uploaded project on the PC. This function cannot be used to overwrite an existing project folder.



1.17. Libraries

The workbench provides various methods for sharing UDFBs, sub-programs and data types (structures) among projects:

- Use the Project *Export* and *Import Assistant* commands for exchanging items:

Program, sub-programs, UDFBs and structures can be exported as single files, suffixed with ".XK5". This enables management of a shared folder containing a collection of export files to be imported in later projects. In this way, several .XK5 files for various versions of the same item can be created.

- Use items from another project (external objects) in a project.

The wokbench enables you to use in a projects some objects defined in other projects, such as UDFBs. This feature is available from the "Project / Settings" dialog box. This provides an easy way to manage libraries of IEC written function blocks that can be re-used in various projects. In addition to this feature, the workbench provides an easy way to safely handle externally defined objects in the local project. External objects are copied locally in the project so that the project remains consistent if the library folder disappears or changes.

- Work with true libraries

Alternatively you can create pure "libraries" to be referenced in projects. The link with libraries is automatically performed each time the project is open or re-built. This ensures you that your project always use the latest version of library objects, but it imposes that the library folder is found at build time.



1.17.1. Working with external objects

The workbench enables you to use in a projects some objects defined in other projects, such as UDFBs. This provides an easy way to manage libraries of IEC written function blocks that can be re-used in various projects. In addition to this feature, the workbench provides an easy way to safely handle externally defined objects in the local project.

You can use in your local project some items from one or several external projects. They can be:

- sub-programs
- user defined function blocks (UDFBs)
- structures

When you select some external objects, they are saved locally in the project. This enables safe maintenance of the local project even in case external projects are modified or deleted. The workbench also provides you some comparison tools so that you can check possible modifications of external objects before you update them in your local project.

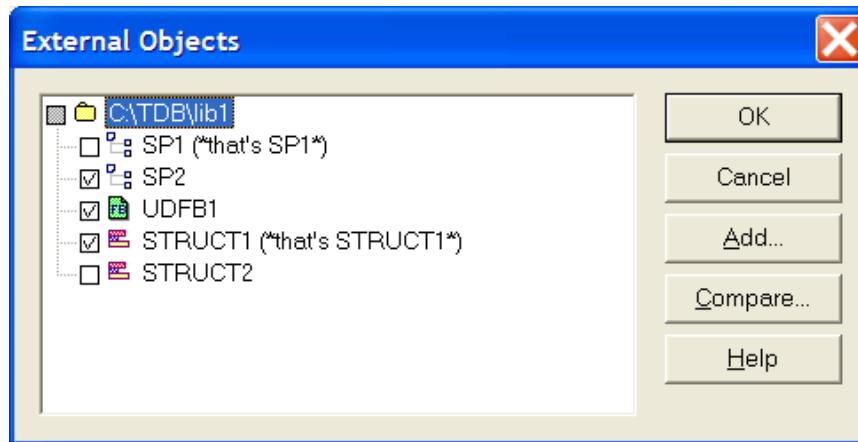
Included external objects cannot be modified in the local Workbench, and are highlighted with a special color in the project workspace. By the way, external objects can be open so that you can explore them during editing or debugging.

1.17.1.1. Using External Objects

Icon Description

- | | |
|--|---|
| | Run the Project / Settings menu command and press the Edit button in the External Objects group of the project settings dialog box. |
|--|---|

The following dialog box is then open and enables you to manage used external objects:



Press the **Add...** button to add in the list an external project. You have to select the folder where this project is located. Then you have to check each object in this external project you want to use in your local project.

Press the **Compare** button to visually compare items that are existing in both in local and external projects. The comparison is performed using the project comparison tool.

Press **OK** to validate the configuration. At this time:



- Any newly checked item is added to the local project.
- Any unchecked item is removed.
- Any checked item already in the project is updated.



Info:

Some items may be marked in red if their name is already used by an item of the local project (in that case, using it is impossible).

It may happen that some external items previously added to the local project are no more existing in the external project. If you uncheck such items in the list, they will be definitively removed when pressing **OK**. If you keep them checked, they remain in the local project as they are so that you can still safely build your project.

It may also happen that a complete external project previously used is no more found on the disk. All used items are kept in the local project unless you uncheck them.



1.17.2. Libraries of sub-programs and UDFBs

The workbench enables you to create pure "libraries" of sub-programs, UDFBs and data structures to be referenced in projects. The link with libraries is automatically performed each time the project is open or re-built. This ensures you that your project always uses the latest version of library objects, but it imposes that the library folder is found at build time.



Alternatively, you can use other methods for sharing programs and sub-programs.

1.17.2.1. What is a library ?

A "library" is similar to a project, but has a special "mark" so that it can be used as a library. All sub-programs, UDFBs and data structures of a library can be used in projects referring to it. Additionally, the library can be open as a normal project, and may contain other material (main programs, global variables...) in order for the library designer to test and validate locally the shared objects of the library.

1.17.2.2. Creating a library

To create a library, use the File / New Project command, and select the *Library* choice in the upper side list of the project creation wizard. The Workbench uses different color settings when editing a library. You can also use the File / Save Project As menu command to save an existing project as a library or the opposite.

1.17.2.3. Linking a project to libraries

In order to use library items in a project, you must link it to the library. For that run the Project / Libraries menu command of the Workbench when editing the project. A box is open and shows the list of already linked libraries. Press the **Add** button to browse the disk for a library folder to be used in the project. Press the **Remove** button to unlink the project from the library selected in the list. Libraries are checked and scanned when you close the box.

A project can be linked to several libraries.

Library items (sub-programs and UDFBs) are not visible in the Workspace area, but can be selected from the bottom-right list area in the Workbench.

Any sub-program or UDFB of a linked library can be used in a program as any other block. The definition of the library blocks (their inputs and outputs) is refreshed automatically:

- When the project is open.
- Before the project is build.
- When you close the library selection box (even with no change).



If the project refers to a library that actually does not exist, an error message is displayed. Compiling errors will happen in that case if you use in your programs items of the missing library.



Info:

A library can be linked to another library. But in that case, when used in a project, the project must be linked to both of them.

Tree	Description
Project1	"Project1" must be linked to "LIB1" and LIB2"
uses LIB1	"LIB1" must be linked to "LIB2"
uses LIB2	

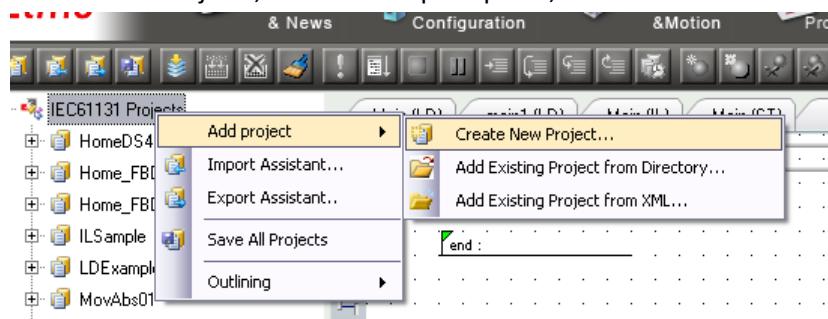
You can never open the contents (source code) of a library item from a project. When debugging, you cannot step into a UDFB or sub-program from a library. Library items are not scanned during a "Find in files" research.



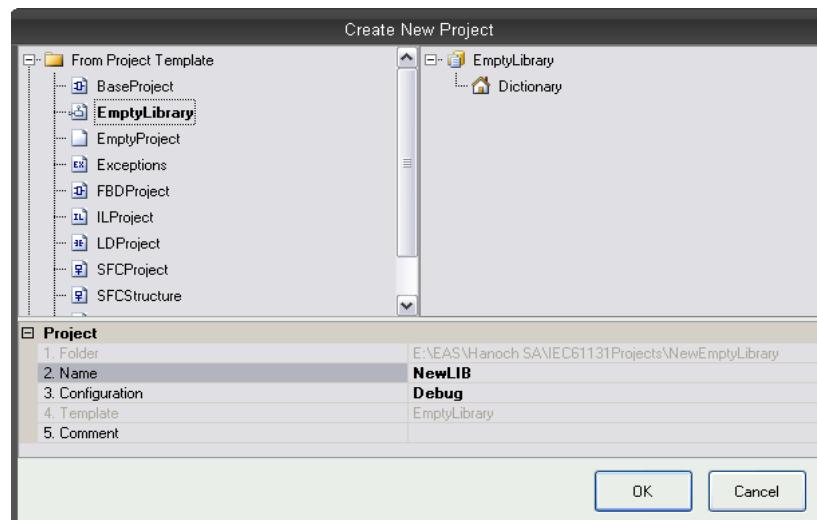
1.17.3. Creating a User Library

To create a user library:

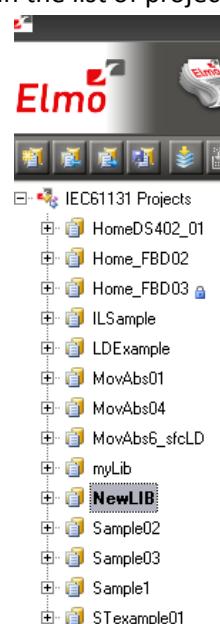
1. Right-click on IEC61131 Projects, in the Workspace panel, and select **Create New Project**.



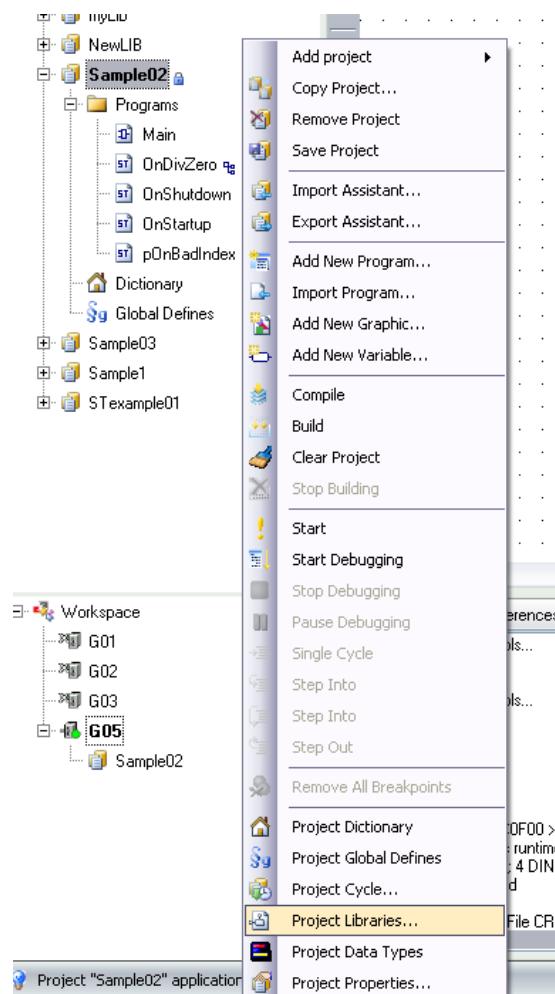
2. From the list of Project Templates, select **Empty Library**, and enter the library name.



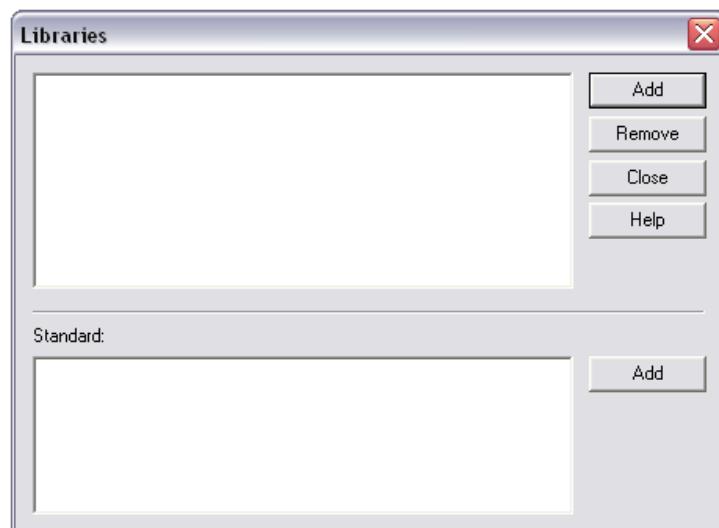
3. Click **OK**. The Library name appears in the list of projects under IEC61131 projects.



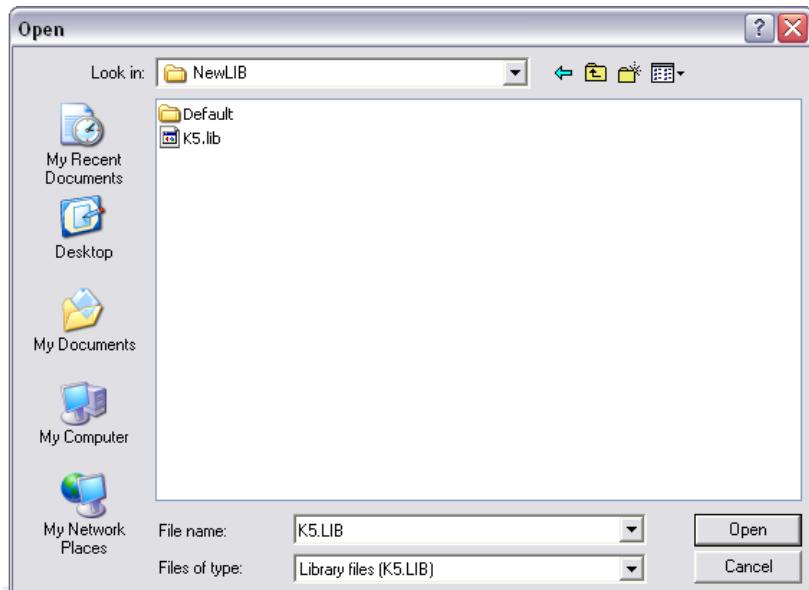
4. Right-click on the new library (default name NewLIB), and select **Project Libraries**.



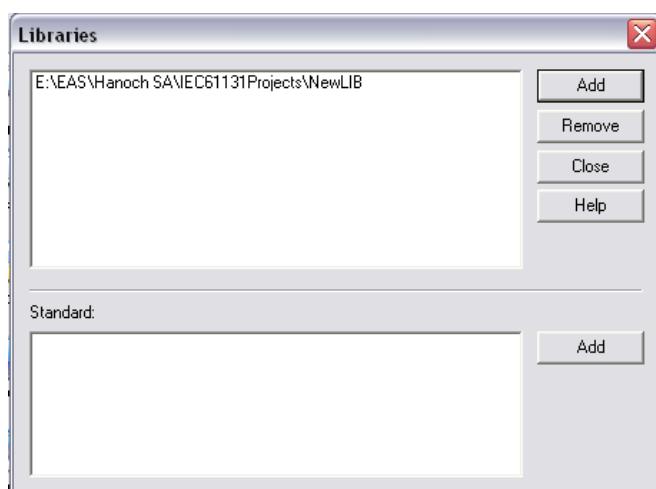
The Libraries window opens.



5. Select the top option **Add**, to add a library, and locate the new library file in the IEC61131 Projects directory.



6. Click **Open**. The library is displayed in the Libraries windows. This displays then project updated libraries.



7. Click **Close** to close the Libraries window.



Chapter 2: Programming languages - Reference guide

Refer to the following pages for an overview of the IEC61131-3 programming languages:

Program organization units	Data types
Variables	Arrays
Constant expressions	Conditional compiling
Handling exceptions	Variable status bits
SFC: Sequential Function Chart	FBD: Function Block Diagram
LD: Ladder Diagram	ST: Structured Text
IL: Instruction List	Use of ST instructions in graphic languages

The following topics detail the set of programming features and standard blocks:

Basic operations	Boolean operations
Arithmetic operations	Comparisons
Type conversion functions	Selectors
Registers	Counters
Timers	Maths
Trigonometrics	String operations
Advanced	

 Info

**Some other functions not documented here are reserved for diagnostics and special operations.
Please contact your technical support for further information.**



2.1. Program organization units

An application is a list of programs. Programs are executed sequentially within the target cycle, according to the following model:

```
Begin cycle
| exchange I/Os
| execute first program
| ...
| execute last program
| wait for cycle time to be elapsed
End Cycle
```

Programs are executed according to the order defined by the user. All SFC programs must be grouped (it is not possible to insert a program in FBD, LD, ST or IL in between two SFC programs). The number of programs in an application is limited to 32767. Each program is entered using a language chosen when the program is created. Possible languages are Sequential Function Chart (SFC), Function Block Diagram (FBD), Ladder Diagram (LD), Structured Text (ST) or Instruction List (IL).

Programs must have unique names. The name cannot be a reserved keyword of the programming languages and cannot have the same name as a standard or "C" function or function block. A program should not have the same name as a declared variable. The name of a program should begin by a letter or an underscore ("_") mark, followed by letters, digits or underscore marks. It is not allowed to put two consecutive underscores within a name. Naming is case insensitive. Two names with different cases are considered as the same.

2.1.1. Child SFC programs

You can define a hierarchy of SFC programs, entered as a tree in the list of programs. A child program is controlled within action blocks of the parent SFC program.

2.1.2. User Defined Function Blocks

The list of programs may be completed by *User Defined Function Blocks* (UDFBs). UDFBs are described using SFC, FBD, LD, ST or IL language, and can be used as other function blocks in the programs of the application. Input and output parameters plus private variables of a UDFB are declared in the variable editor as local variables of the UDFB.

There is no restriction using any operation in a UDFB. A UDFB can call standard functions and function blocks.

A UDFB can call another UDFB. The called UDFB must be declared before the calling one in the program list.

Each time a UDFB is instantiated, its private variables are duplicated for the declared instance. The code of the UDFB is duplicated on each call in parent programs. This leads to higher performances at run time, but consumes code space. It is advised recommended to package small algorithms in UDFBs. Large parts of code should be managed in programs.

A UDFB cannot have more than 32 input parameters or 32 output parameters.



2.1.3. Sub-programs

The list of programs may be completed by *Sub-programs*. Sub-programs are described using FBD, LD, ST or IL language, and can be called by the programs of the application. Input and output parameters plus local variables of a sub-program are declared in the variable editor as local variables of the sub-program.

A sub-program may call another sub-program or a UDFB.

Unlike UDFB, local variables of a sub program are not instantiated. This means that the sub-program always work on the same set of local variables. Local variables of a sub-program keep their value among various calls. The code of a sub-program is not duplicated when called several times by parent programs.

A sub-program cannot have more than 32 input parameters or 32 output parameters.



2.2. Data types

2.2.1. Basic data types

Below are the available basic data types:

Type	Description
BOOL	Boolean (bit) - can be <i>FALSE</i> or <i>TRUE</i> - stored on 1 bit
SINT (*)	Small signed integer on 8 bits (from -128 to +127)
USINT (*)	Small unsigned integer on 8 bits (from 0 to +255)
BYTE	Same as USINT
INT (*)	Signed integer on 16 bits (from -32768 to +32767)
UINT (*)	Unsigned integer on 16 bits (from 0 to +65535)
WORD	Same as UINT
DINT	Signed integer on 32 bits (from -2147483648 to +2147483647)
UDINT (*)	Unsigned integer on 32 bits (from 0 to +4294967295)
DWORD	Same as UDINT
LINT (*)	Long signed integer on 64 bits
REAL (*)	Single precision floating point - stored on 32 bits
LREAL (*)	Double precision floating point - stored on 64 bits
TIME	Time of day - less than 24h - accuracy is 1ms
STRING (*)	Variable length string with declared maximum length. The declared maximum length cannot exceed 255 characters.

(*) Some of those data types may be not supported by all targets.

2.2.2. Structures

A structure is a complex data type defined as a set of members. Members of a structure may have various data types. A member of a structure may have dimensions or may be an instance of another structure.

When a structure is defined, it may be used as other data types to declare variables.

Members of a structure may have an initial value. In that case, corresponding members of all declared variable having this structure type will be initialized with the initial value of the member.

For specifying a member of a structured variable in languages, use the following notation:

VariableName.MemberName



2.2.3. Enumerated data types

You can define some new data types that are enumeration of named values. For Example:

type: LIGHT

values: GREEN, ORANGE, RED

Then in programs, you can use one of the enumerated values, prefixed by the type name:

Light1 := LIGHT#RED;

Variables having enumerated data types can only be used for assignment, comparison, and SEL/MUX functions.

2.2.4. "Bit fields" data types

You can define new data types derived from integer data types, that have some readable names for some of their bits. Thus you can use VarName.BitName notations in programs. Such data types cannot be derived from the LINT type.



2.3. Variables

All variables used in programs must be first declared in the variable editor. Each variable belongs to a group and is must be identified by a unique name within its group.

2.3.1. Groups

A group is a set of variables. A group either refers to a physical class of variables, or identifies the variables local to a program or user defined function block. Below are the possible groups:

Group	Description
GLOBAL	Internal variables known by all programs.
RETAIN	Non volatile internal variables known by all programs.
%I...	Channels of an input board - variables with same data type linked to a physical input device.
%Q...	Channels of an output board - variables with same data type linked to a physical output device.
PROGRAMxxx	All internal variables local to a program. The name of the group is the name of the program.
UDFBxxx	All internal variables local to a User Defined Function Block plus its IN and OUT parameters. The name of the group is the name of the program.

2.3.2. Data type and dimension

Each variable must have a valid data type. It can be either a basic data type or a function block. In that case the variable is an instance of the function block. Physical I/Os must have a basic data type. Instances of function blocks can refer either to a standard or "C" embedded block, or to a User Defined Function Block.

If the selected data type is STRING, you must specify a maximum length, that cannot exceed 255 characters.

Refer to the list of available data types for more information. Refer to the section describing function blocks for further information about how to use a function instance.

Additionally, you can specify dimension(s) for an internal variable, in order to declare an array. Arrays have at most 3 dimensions. All indexes are 0 based. For instance, in case of single dimension array, the first element is always identified by *Array Name[0]*. The total number of items in an array (merging all dimensions) cannot exceed 65535.



2.3.3. Naming a variable

A variable must be identified by a unique name within its parent group. The variable name cannot be a reserved keyword of the programming languages and cannot have the same name as a standard or "C" function or function block. A variable should not have the same name as a program or a user defined function block.

The name of a variable should begin by a letter or an underscore ("_") mark, followed by letters, digits or underscore marks. It is not allowed to put two consecutive underscores within a variable name. Naming is case insensitive. Two names with different cases are considered as the same.

2.3.4. Naming Physical I/Os

Each I/O channel has a predefined symbol that reflects its physical location. This symbol begins with %*I* for an input and %*Q* for an output, followed by a letter identifying the physical size of the data. Then comes the location of the board, expressed on 1 or two numbers, and finally the 0 based index of the channel within the board. All numbers are separated by dots. Below are the possible prefixes for IO symbols:

Prefix	Description
%IX	1 byte input - BOOL or SINT
%QX	1 byte output - BOOL or SINT
%IW	2 bytes input - INT
%QW	2 bytes output - INT
%ID	4 bytes input - DINT or REAL
%QD	4 bytes input - DINT or REAL
%IL	8 bytes input - LINT or LEAL
%QL	8 bytes output - LINT or LEAL
%IS	STRING input
%QS	STRING output

Additionally, you can give an alias (a readable name) to each I/O channel. In that case, either the "%" name or the alias can be used in programs with no difference. The alias must fit to the same rules as a variable name.

2.3.5. Attributes of a variable

Physical I/Os are marked as either *Input* or *Output*. Inputs are read-only variables. For each internal variable, you can select the *Read Only*.

Parameters of User Defined Function Blocks and sub-programs are marked as either *IN* or *OUT*.



2.3.6. Parameters of sub-programs and UDFBs

Sub-programs and UDFBs may have parameters on input or output. Output parameters cannot be arrays of data structures but only single data. When an array is passed as an input parameter to a UDFB, it is considered as *INOUT* so the UDFB can read or write in it. The support of complex data types for input parameters may depend on selected compiling options.



2.4. Arrays

You can specify dimension(s) for internal variables, in order to declare arrays. All indexes are 0 based. For instance, in case of single dimension array, the first element is always identified by *ArrayName[0]*.

To declare an array, enter its dimension in the corresponding column of the variable editor. For a multi-dimension array, enter dimensions separated by commas (ex: 2,10,4).

2.4.1. Use in ST and IL Languages

To specify an item of an array in ST and IL language, enter the name of the array followed by the index(es) entered between "[" and "]" characters. For multi-dimension arrays, enter indexes separated by commas. Indexes may be either constant or complex expressions.



Example

```
TheArray[1,7] := value;  
result := SingleArray[i + 2];
```

2.4.2. Use in FBD and LD Languages

In graphical languages, the following blocks are available for managing array elements:

Block	Description
[I]>>	Get value of an item in a single dimension array.
[I,J]>>	Get value of an item in a two dimension array.
[I,J,K]>>	Get value of an item in a three dimension array.
>>[I]	Set value of an item in a single dimension array.
>>[I,J]	Set value of an item in a two dimension array.
>>[I,J,K]	Set value of an item in a three dimension array.

For *get* blocks, the first input is the array and the output is the value of the item. Other inputs are indexes in the array.

For *put* blocks, the first input is the forced value and the second input is the array. Other inputs are indexes in the array.



Attention

- Arrays have at most 3 dimensions.
- All indexes are 0 based.
- The total number of items in an array (merging all dimensions) cannot exceed 65535.



2.5. Constant expressions

Constant expressions can be used in all languages for assigning a variable with a value. All constant expressions have a well defined data type according to their semantics. If you program an operation between variables and constant expressions having inconsistent data types, it will lead to syntactic errors when the program is compiled. Below are the syntactic rules for constant expressions according to possible data types:

Expression	Type	Details
BOOL	Boolean	There are only two possible Boolean constant expressions. They are reserved keywords <i>TRUE</i> and <i>FALSE</i> .
SINT	Small (8 bit) Integer	Small integer constant expressions are valid integer values (between -128 and 127) and must be prefixed with <i>SINT#</i> . All integer expressions having no prefix are considered as DINT integers.
USINT / BYTE	Unsigned 8 bit Integer	Unsigned small integer constant expressions are valid integer values (between 0 and 255) and must be prefixed with <i>USINT#</i> . All integer expressions having no prefix are considered as DINT integers.
INT	16 bit integer	16 bit integer constant expressions are valid integer values (between -32768 and 32767) and must be prefixed with <i>INT#</i> . All integer expressions having no prefix are considered as DINT integers.
UINT / WORD	Unsigned 16 bit integer	Unsigned 16 bit integer constant expressions are valid integer values (between 0 and 255) and must be prefixed with <i>UINT#</i> . All integer expressions having no prefix are considered as DINT integers.
DINT	32 bit (default) integer	32 bit integer constant expressions must be valid numbers between -2147483648 to +2147483647. DINT is the default size for integers: such constant expressions do not need any prefix. You can use <i>2#</i> , <i>8#</i> or <i>16#</i> prefixes for specifying a number in respectively binary, octal or hexadecimal basis.
UDINT / DWORD	Unsigned 32 bit integer	Unsigned 32 bit integer constant expressions are valid integer values (between 0 and 4294967295) and must be prefixed with <i>UDINT#</i> . All integer expressions having no prefix are considered as DINT integers.
LINT	Long (64 bit) integer	Long integer constant expressions are valid integer values and must be prefixed with <i>LINT#</i> . All integer expressions having no prefix are considered as DINT integers.
REAL	Single precision floating point value	Real constant expressions must be valid number, and must include a dot ("."). If you need to enter a real expression having an integer value, add <i>.0</i> at the end of the number. You can use <i>F</i> or <i>E</i> separators for specifying the exponent in case of a scientist representation. REAL is the default precision for floating points: such expressions do not need



Expression	Type	Details	
		any prefix.	
LREAL	Double precision floating point value	Real constant expressions must be valid number, and must include a dot ("."), and must be prefixed with <i>LREAL#</i> . If you need to enter a real expression having an integer value, add .0 at the end of the number. You can use <i>F</i> or <i>E</i> separators for specifying the exponent in case of a scientist representation.	
TIME	Time of day	Time constant expressions represent durations that must be less than 24 hours. Expressions must be prefixed by either <i>TIME#</i> or <i>T#</i> . They are expressed as a number of hours followed by <i>h</i> , a number of minutes followed by <i>m</i> , a number of seconds followed by <i>s</i> , and a number of milliseconds followed by <i>ms</i> . The order of units (hour, minutes, seconds, milliseconds) must be respected. You cannot insert blank characters in the time expression. There must be at least one valid unit letter in the expression.	
STRING	Character string	String expressions must be written between single quote marks. The length of the string cannot exceed 255 characters. You can use the following sequences to represent a special or not printable character within a string:	
		Sequence	Description
		\$\$	a "\$" character
		\$'	a single quote
		\$T	a tab stop (ASCII code 9)
		\$R	a carriage return character (ASCII code 13)
		\$L	a line feed character (ASCII code 10)
		\$N	carriage return plus line feed characters (ASCII codes 13 and 10)
		\$P	a page break character (ASCII code 12)
		\$xx	any character (xx is the ASCII code expressed on two hexadecimal digits)



Example

Below are some Examples of valid constant expressions:

Expression	Description
TRUE	TRUE Boolean expression



Expression	Description
FALSE	FALSE Boolean expression
SINT#127	small integer
INT#2000	16 bit integer
123456	DINT (32 bit) integer
16#abcd	DINT integer in hexadecimal basis
LINT#1	long (64 bit) integer having the value "1"
0.0	0 expressed as a REAL number
1.002E3	1002 expressed as a REAL number in scientist format
LREAL#1E-200	Double precision real number
T#23h59m59s999ms	maximum TIME value
TIME#0s	null TIME value
T#1h123ms	TIME value with some units missing
'hello'	character string
'name\$Tage'	character string with two words separated by a tab
'I\$m here'	character string with a quote inside (I'm here)
'x\$00y'	character string with two characters separated by a null character (ASCII code 0)

Below are some Examples of typical errors in constant expressions:

Expression	Error-Description
BooVar := 1; 1a2b 1E-200 T#12 'I'm here' hello	0 and 1 cannot be used for Booleans basis prefix ("16#") omitted "LREAL#" prefix omitted for a double precision float Time unit missing quote within a string with "\$" mark omitted quotes omitted around a character string



2.6. Conditional compiling

The compiler supports conditional compiling directives in ST, IL, LD, and FBD languages. Conditional compiling directives condition the inclusion of a part of the program in the generated code. Conditional compiling is an easy way to manage several various configurations and options in a unique application programming.

Conditional compiling uses definitions as conditions. Below is the main syntax:

```
#ifdef CONDITION
    statementsYES...
#else
    statementsNO...
#endif
```

If CONDITION has been defined using #define syntax, then the *statementsYES* part is included in the code, else the *statementsNO* part is included. The *#else* statement is optional.

In ST and IL text languages, directives must be entered alone on one line of text. In FBD language, directives must be entered as the text of network breaks. In LD language, directives must be entered on comment lines.

The condition *_DEBUG* is automatically defined when the application is compiled in *DEBUG* mode. This allows you to incorporate some additional statements (such as trace outputs) in your code that are not included in *RELEASE* mode.



2.7. Handling exceptions

The compiler enables you to write your own exception programs for handling particular system events. The following exceptions can be handled:

- Startup (before the first cycle)
- Shutdown (after the last cycle)
- Division by zero

2.7.1. Startup

You can write your own exception program to be executed before the first application cycle is executed:

1. Create a new main program that will handle the exception. It cannot be a SFC program.

In the editor of global defines, insert the following line:

#OnStartup ProgramName



The program is executed before all other programs within the first cycle. This implies that the cycle timing may be longer during the first cycle. You cannot put breakpoints in the Startup program.

2.7.2. Shutdown

You can write your own exception program to be executed after the last application cycle when the runtime system is cleanly stopped:

1. Create a new main program that will handle the exception. It cannot be a SFC program.

In the editor of global defines, insert the following line:

#OnShutdown ProgramName



You cannot put breakpoints in the Shutdown program.

2.7.3. Division by zero

You can write your own exception program for handling the "Division by zero" exception. Below is the procedure you must follow for setting an exception handler:

1. Create a new sub-program without any parameter that will handle the exception

In the editor of global defines, insert the following line:

#OnDivZero SubProgramName

In the sub-program that handles the exception you can perform any safety or trace operation.



You then have the selection between the following possibilities:

- Return without any special call. In that case the standard handling will be performed: a system error message is generated, the result of the division is replaced by a maximum value and the application continues.
- Call the FatalStop function. The runtime then stops immediately in Fatal Error mode.
- Call the CycleStop function. The runtime finishes the current program and then turns in *cycle setting* mode.

Handlers can also be used in *DEBUG* mode for tracking the bad operation. Just put a breakpoint in your handler. When stopped, the call stack will show you the location of the division in the source code of the program.

2.7.4. Array index out of bounds

You can write your own exception program for handling the "Array index out of bounds" exception.

Below is the procedure you must follow for setting an exception handler:

1. Create a new sub-program without any parameter that will handle the exception

In the editor of global defines, insert the following line:

`#OnBadArrayIndex SubProgramName`



Attention

This is anyway a fatal error. If the "Check array bounds" compiling option is set, the runtime goes in "fatal error" mode after calling your sub-program.



2.8. Variable status bits

The workbench enables you to associate status bits to declared variables. Each variable may have, in addition to its real time value:

- 64 status bits
- a date and time stamp

Status bits and time stamps are generally set by input drivers taking care of hardware inputs, but may also be transported together with the value of the variable on some network protocols. In addition, the IEC 61131-3 programs may access to the status bits of variables.



Attention

- Status bit management may be not available on some targets. Please refer to OEM instructions for further details about available features.
- Status bit management is CPU and memory consuming and may reduce the performances of your applications.

2.8.1. Enabling status bits

In order to enable the management of status bits and time/date stamps by the runtime, you must check the following option in the list of compiler options from the Project Settings wizard:

Allocate status flags for variables with embedded properties

Only variables having some properties defined (either a profile attached or embedded symbol) will get status bits. Status bits are available only for global scope variables (global, retain, IOs...) with a single data type (cannot be array or structure).

2.8.2. Reading and writing status from programs

The following functions are available for managing status Information in the programs:

<i>vsiGetBit</i>	Get a status bit of a variable
<i>vsiGetDate</i>	Get the date stamp of a variable
<i>vsigetTime</i>	Get the time stamp of a variable
<i>vsiSetBit</i>	Set a status bit of a variable
<i>vsiSetDate</i>	Set the date stamp of a variable
<i>vsiSetTime</i>	Set the time stamp of a variable
<i>vsiStamp</i>	Update the stamp of a variable according to the current time



Below is the syntax of the functions:

```
bBit := vsiGetBit ( variable, bitID );
iDate := vsiGetDate ( variable );
iTime := vsiGetTime ( variable );
bOK := vsiSetBit ( variable, bitID, bBit );
bOK := vsiSetDate ( variable, iDate );
bOK := vsiSetTime ( variable, iTime );
bOK := vsiStamp ( variable );
```

The functions use the following arguments:

Argument	Description
variable	Variable having embedded profile or symbol.
bitID : DINT	ID of a status bit (see list of IDs in the section below).
bBit : BOOL	Value of the status bit.
iDate : DINT	Date stamp according to real time clock functions conventions.
iTime : DINT	Time stamp according to real time clock functions conventions.
bOK : BOOL	<i>TRUE</i> if successful.

See the description of real time clock functions for further information about time and date stamps.

2.8.3. Drivers supporting status bits

Below are runtime drivers taking care of status bits and date/time stamping:

Driver	Description
Variable binding (ETHERNET)	Binding (spontaneous protocol) is used for real time exchange of variable values among runtimes over ETHERNET. The protocol takes care of carrying status bits. The protocol updates the date and time stamps of variables updated by the network.
MODBUS Master	The MODBUS master protocols (RTU / TCP / UDP) takes care of updating the date and time stamp of all variables updated by the network. The MODBUS stack also sets the _VSB_I_BIT status bits of received variables according to the exchange error status.
MODBUS Slave	The MODBUS slave protocols (RTU / TCP / UDP) takes care of updating the date and time stamp of all variables updated by the network.
IEC 60870-5 Slave	The IEC 60870-5-101 and IEC 60870-5-104 slave protocols send the _VSB_I_BIT, _VSB_OV_BIT, _VSB_BL_BIT, _VSB_SP_BIT and _VSB_NT_BIT in the protocol telegrams for points and measures. Update of date/time stamp included.



Driver	Description
IEC 61850 Server	Variable status bits are not supported by the IEC 61850 Server.
IEC 61850 Client	The IEC 61850 Client can read the _VSB_I_BIT from the IEC 61850 Server. Update of date/time stamp included.
zenon RT to straton connection	The zenon RT to straton connection can read all 64 status bits. Update of date/time stamp included.

2.8.4. List of status bits

Below is the list of available status bits. Identifiers (_VSB_...) are predefined in the compiler and can be directly used in the programs:

bit	identifier	description
0	_VSB_ST_M1	user defined status
1	_VSB_ST_M2	user defined status
2	_VSB_ST_M3	user defined status
3	_VSB_ST_M4	user defined status
4	_VSB_ST_M5	user defined status
5	_VSB_ST_M6	user defined status
6	_VSB_ST_M7	user defined status
7	_VSB_ST_M8	user defined status
8	_VSB_SELEC	Select
9	_VSB_REV	Revision
10	_VSB_DIREC	Desired direction
11	_VSB_RTE	Runtime exceeded
12	_VSB_MVALUE	Manual value
13	_VSB_ST_14	user defined status
14	_VSB_ST_15	user defined status
15	_VSB_ST_16	user defined status
16	_VSB_GR	General request
17	_VSB_SPONT	Spontaneous
18	_VSB_I_BIT	Invalid



bit	identifier	description
19	_VSB_SUWI	Summer/Winter time announcement
20	_VSB_N_UPD	Switched off
21	_VSB_RT_E	Realtime external
22	_VSB_RT_I	Realtime internal
23	_VSB_NSORT	Not sortable
24	_VSB_DM_TR	Default message trafo value
25	_VSB_RM_TR	Run message trafo value
26	_VSB_INFO	Info for variable
27	_VSB_AVALUE	Alternative value
28	_VSB_RES28	reserved
29	_VSB_ACTUAL	Not updated
30	_VSB_WINTER	Winter time
31	_VSB_RES31	reserved
32	_VSB_TCB0	Transmission cause
33	_VSB_TCB1	Transmission cause
34	_VSB_TCB2	Transmission cause
35	_VSB_TCB3	Transmission cause
36	_VSB_TCB4	Transmission cause
37	_VSB_TCB5	Transmission cause
38	_VSB_PN_BIT	P/N bit
39	_VSB_T_BIT	Test bit
40	_VSB_WR_ACK	Acknoledge writing
41	_VSB_WR_SUC	Writing successful
42	_VSB_NORM	Normal status
43	_VSB_ABNORM	Deviation normal status
44	_VSB_BL_BIT	IEC status: blocked



bit	identifier	description
45	_VSB_SP_BIT	IEC status: substituted
46	_VSB_NT_BIT	IEC status: not typical
47	_VSB_OV_BIT	IEC status: overflow
48	_VSB_SE_BIT	IEC status: select
49	not defined	
50	not defined	
51	not defined	
52	not defined	
53	not defined	
54	not defined	
55	not defined	
56	not defined	
57	not defined	
58	not defined	
59	not defined	
60	not defined	
61	not defined	
62	not defined	
63	not defined	



2.9. Basic Operations

Below are the language features for basic data manipulation:

- Variable assignment
- Bit access
- Parenthesis
- Calling a function
- Calling a function block
- Calling a sub-program
- MOVEBLOCK: Copying/moving array items
- COUNTOF: Number of items in an array
- INC: Increase a variable
- DEC: decrease a variable

Below are the language features for controlling the execution of a program:

- Labels
- Jumps
- RETURN

Below are the structured statements for controlling the execution of a program:

Statement	Description
IF	Conditional execution of statements
WHILE	Repeat statements while a condition is <i>TRUE</i> .
REPEAT	Repeat statements until a condition is <i>TRUE</i>
FOR	Execute iterations of statements
CASE	Switch to one of various possible statements
EXIT	Exit from a loop instruction
WAIT	Delay program execution
ON	Conditional execution



2.9.1. Assignment

Function `:= LD LDN ST STN`

Operator - variable assignment

Inputs IN : ANY Any variable or complex expression

Outputs Q : ANY Forced variable

Diagram

Remarks The output variable and the input expression must have the same type. The forced variable cannot have the read only attribute.

In LD and FBD languages, the 1 block is available to perform a "1 gain" data copy.
In LD language, the input rung (EN) enables the assignment, and the output rung keeps the state of the input rung.

In IL language, the LD instruction loads the first operand, and the ST instruction stores the current result into a variable. The current result and the operand of ST must have the same type. Both LD and ST instructions can be modified by N in case of a Boolean operand for performing a Boolean negation.

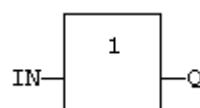
ST Language `Q := IN; (* copy IN into variable Q *)`

`Q := (IN1 + (IN2 / IN 3)) * IN4; (* assign the result of a complex expression *)`

`result := SIN (angle); (* assign a variable with the result of a function *)`

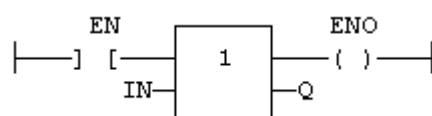
`time := MyTon.ET; (* assign a variable with an output parameter of a function block *)`

FBD Language



LD Language

The copy is executed only if EN is TRUE. ENO has the same value as EN.



IL Language

`Op1: LD IN (* current result is: IN *)`

`ST Q (* Q is: IN *)`

`LDN IN1 (* current result is: NOT (IN1) *)`

`ST Q (* Q is: NOT (IN1) *)`

`LD IN2 (* current result is: IN2 *)`

`STN Q (* Q is: NOT (IN2) *)`



2.9.2. Access to bits of an integer

You can directly specify a bit within n integer variable in expressions and diagrams, using the notation:

Variable.BitNo

Where the **Variable** is the name of an integer variable, and **BitNo** is the number of the bit in the integer.

The variable can have one of the following data types, where 0 always represents the less significant bit:

SINT, USINT, BYTE (8 bits from .0 to .7)
INT, UINT, WORD (16 bits from .0 to .15)
DINT, UDINT, DWORD (32 bits from .0 to 31)
LINT (from 0 to 63)

2.9.3. Calling a function

A function calculates a result according to the current value of its inputs. Unlike a function block, a function has no internal data and is not linked to declared instances. A function has only one output: the result of the function. A function can be:

- A standard function (SHL, SIN...).
- A function written in "C" language and embedded on the target.

ST Language

To call a function block in ST, you have to enter its name, followed by the input parameters written between parenthesis and separated by commas. The function call may be inserted into any complex expression. A function call can be used as an input parameter of another function. The following Example demonstrates a call to ODD and SEL functions:



Example

```
(* The following statement converts any odd integer  
value into the nearest even integer: *)
```

```
iEvenVal := SEL( ODD( iValue ), iValue, iValue+1 );
```

FBD and LD Languages

To call a function block in FBD or LD languages, you just need to insert the function in the diagram and to connect its inputs and output.

IL Language

To call a function block in IL language, you must load its first input parameter before the call, and then use the function name as an instruction, followed by the other input parameters, separated by commas. The result of the function is then the current result. The following Example demonstrates a call to ODD and SEL functions:



Example

```
(* The following statement converts any odd integer  
into 0: *)
```

```
Op1: LD iValue ODD SEL iValue, 0 ST iResult
```



2.9.4. Calling a function block

CAL**CALC****CALNC****CALCN**

A function block groups an algorithm and a set of private data. It has inputs and outputs. A function block can be:

- A standard function block (RS, TON...).
- A block written in "C" language and embedded on the target.
- A User Defined Function Block (UDFB) written in ST, FBD, LD or IL.

To use a function block, you have to declare an instance of the block as a variable, identified by a unique name. Each instance of a function block as its own set of private data and can be called separately. A call to a function block instance processes the block algorithm on the private data of the instance, using the specified input parameters.

ST Language

To call a function block in ST, you have to specify the name of the instance, followed by the input parameters written between parenthesis and separated by commas. To have access to an output parameter, use the name of the instance followed by a dot '.' and the name of the desired parameter. The following Example demonstrates a call to an instance of TON function block (MyTimer is declared as an instance of TON):



Example

```
MyTimer (bTrig, t#2s );  
TimerOutput := MyTimer.Q;  
ElapsedTime := MyTimer.ET;
```

FBD and LD Languages

To call a function block in FBD or LD languages, you just need to insert the block in the diagram and to connect its inputs and outputs. The name of the instance must be specified upon the rectangle of the block.

IL Language

To call a function block in IL language, you must use the CAL instruction, and use a declared instance of the function block. The instance name is the operand of the CAL instruction, followed by the input parameters written between parenthesis and separated by commas. Alternatively the CALC, CALNC or CALCN conditional instructions can be used:

CAL Calls the function block.

CALC Calls the function block if the current result is TRUE.

CALNC Calls the function block if the current result is FALSE.

CALCN same as CALNC.

The following Example demonstrates a call to an instance of *TON* function block (MyTimer is declared as an instance of *TON*):



Example

```
Op1: CAL MyTimer (bTrig, t#2s )  
LD MyTimer.Q
```



ST TimerOutput

LD MyTimer.ET

ST ElapsedTimer

Op2: LD bCond

CALC MyTimer (bTrig, t#2s) (* called only if bCond is TRUE *)

Op3: LD bCond

CALNC MyTimer (bTrig, t#2s) (* called only if bCond is FALSE *)



2.9.5. Calling a sub-program

A sub-program is called by another program. Unlike function blocks, local variables of a sub-program are not instantiated, and thus you do not need to declare instances. A call to a sub-program processes the block algorithm using the specified input parameters. Output parameters can then be accessed.

ST Language

To call a sub-program in ST, you have to specify its name, followed by the input parameters written between parenthesis and separated by commas. To have access to an output parameter, use the name of the sub-program followed by a dot '.' and the name of the desired parameter:

```
MySubProg (i1, i2); (* calls the sub-program *)  
Res1 := MySubProg.Q1;  
Res2 := MySubProg.Q2;
```

Alternatively, if a sub-program has one and only one output parameter, it can be called as a function in ST language:

```
Res := MySubProg (i1, i2);
```

FBD and LD Languages

To call a sub-program in FBD or LD languages, you just need to insert the block in the diagram and to connect its inputs and outputs.

IL Language

To call a sub-program in IL language, you must use the CAL instruction with the name of the sub-program, followed by the input parameters written between parenthesis and separated by commas. Alternatively the CALC, CALCN or CALNC conditional instructions can be used:

CAL Calls the sub-program.
CALC Calls the sub-program if the current result is TRUE.
CALNC Calls the sub-program if the current result is FALSE.
CALCN same as CALNC.



Example

```
Op1: CAL MySubProg (i1, i2 )  
      LD  MySubProg.Q1  
      ST  Res1  
      LD  MySubProg.Q2  
      ST  Res2
```



2.9.6. CASE OF ELSE END_CASE

Statement Switch between enumerated statements

Syntax

```
CASE <DINT expression> OF
  <value> :
    <statements>
  <value> , <value> :
    <statements>;
  <value> .. <value> :
    <statements>;
ELSE
  <statements>
END_CASE;
```

Remarks All enumerated values correspond to the evaluation of the DINT expression and are possible cases in the execution of the statements. The statements specified after the ELSE keyword are executed if the expression takes a value that is not enumerated in the switch. For each case, you must specify either a value, or a list of possible values separated by commas (",") or a range of values specified by a "min .. max" interval. You must enter space characters before and after the ".." separator.

ST Language

Example

This Example checks the first prime numbers:

```
CASE iNumber OF
  0 :
    Alarm := TRUE;
    AlarmText := '0 gives no result';
  1 .. 3, 5 :
    bPrime := TRUE;
  4, 6 :
    bPrime := FALSE;
ELSE
  Alarm := TRUE;
  AlarmText := 'I don't know after 6 !';
END_CASE;
```

FBD Language Not available

LD Language Not available

IL Language Not available



See also IF WHILE REPEAT FOR EXIT



2.9.7. CountOf

Function Returns the number of items in an array

Inputs ARR : ANY Declared array

Outputs Q : DINT Total number of items in the array

Remarks The input must be an array and can have any data type. This function is particularly useful to avoid writing directly the actual size of an array in a program, and thus keep the program independent from the declaration.

Example

```
FOR i := 1 TO CountOf (MyArray ) DO
    MyArray[i-1] := 0;
END_FOR;
```

In LD language, the operation is executed only if the input rung (EN) is TRUE. The output rung (ENO) keeps the same value as the input rung.

Example

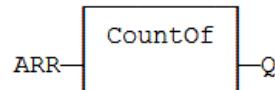
Array	Return
-------	--------

Arr1 [0..9]	10
---------------	----

Arr2 [0..4 , 0..9]	50
----------------------	----

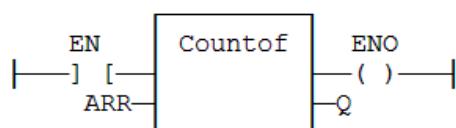
ST Language Q := CountOf (ARR);

FBD Language



LD Language The function is executed only if EN is TRUE.

ENO keeps the same value as EN.



IL Language Not available



2.9.8. DEC

Function	Decrease a numerical variable
Inputs	IN : ANY Numerical variable (increased after call)
Outputs	Q : ANY Decreased value
Remarks	<p>When the function is called, the variable connected to the IN input is decreased and copied to Q. All data types are supported except BOOL and STRING: for these types, the output is the copy of IN.</p> <p>For real values, variable is decreased by 1.0. For time values, variable is decreased by 1ms.</p> <p>The IN input must be directly connected to a variable, and cannot be a constant or complex expression.</p> <p>This function is particularly designed for ST language. It allows simplified writing as assigning the result of the function is not mandatory.</p>
ST Language	<pre>IN := 2; Q := DEC (IN); (* now: IN = 1 ; Q = 1 *) DEC (IN); (* simplified call *)</pre>
FBD Language	
LD Language	
IL Language	Not available



2.9.9. EXIT

Statement Exit from a loop statement

Remarks The *EXIT* statement indicates that the current loop (WHILE, REPEAT or FOR) must be finished. The execution continues after the END_WHILE, END_REPEAT or END_FOR keyword or the loop where the *EXIT* is. *EXIT* quits only one loop and cannot be used to exit at the same time several levels of nested loops.

 **Attention** Loop instructions may lead to infinite loops that block the target cycle.

ST Language This program searches for the first non null item of an array:

```
iFound = -1; (* means: not found *)
FOR iPos := 0 TO (iArrayDim - 1) DO
    IF iPos <> 0 THEN
        iFound := iPos;
        EXIT;
    END_IF;
END_FOR;
```

FBD Language Not available

LD Language Not available

IL Language Not available



2.9.10. FOR TO BY END_FOR

Statement	Iteration of statement execution
Syntax	<pre>FOR <index> := <minimum> TO <maximum> BY <step> DO <statements> END_FOR;</pre> <p>index = DINT internal variable used as index. minimum = DINT expression: initial value for index. maximum = DINT expression: maximum allowed value for index. step = DINT expression: increasing step of index after each iteration (default is 1)</p>
Remarks	The BY <step> statement can be omitted. The default value for the step is 1
ST Language	<pre>iArrayDim := 10; (* resets all items of the array to 0 *) FOR iPos := 0 TO (iArrayDim - 1) DO MyArray[iPos] := 0; END_FOR; (* set all items with odd index to 1 *) FOR iPos := 1 TO 9 BY 2 DO MyArray[iPos] := 1; END_FOR;</pre>
FBD Language	Not available
LD Language	Not available
IL Language	Not available



2.9.11. IF THEN ELSE ELSIF END_IF

Statement Conditional execution of statements

Syntax

```
IF <BOOL expression> THEN
    <statements>
ELSIF <BOOL expression> THEN
    <statements>
ELSE
    <statements>
END_IF;
```

Remarks The IF statement is available in ST only. The execution of the statements is conditioned by a boolean expression. ELSIF and ELSE statements are optional. There can be several ELSIF statements.

ST Language (* simple condition *)

```
IF bCond THEN
    Q1 := IN1;
    Q2 := TRUE;
END_IF;
```

(* binary selection *)

```
IF bCond THEN
    Q1 := IN1;
    Q2 := TRUE;
ELSE
    Q1 := IN2;
    Q2 := FALSE;
END_IF;
```

(* enumerated conditions *)

```
IF bCond1 THEN
    Q1 := IN1;
ELSIF bCond2 THEN
    Q1 := IN2;
ELSIF bCond3 THEN
    Q1 := IN3;
ELSE
    Q1 := IN4;
END_IF;
```

FBD Language Not available

LD Language Not available

IL Language Not available



2.9.12. INC

Function	Increase a numerical variable
Inputs	IN : ANY Numerical variable (increased after call).
Outputs	Q : ANY Increased value.
Remarks	<p>When the function is called, the variable connected to the IN input is increased and copied to Q. All data types are supported except BOOL and STRING: for these types, the output is the copy of IN.</p> <p>For REAL values, variable is increased by 1.0. For TIME values, variable is increased by 1ms.</p> <p>The IN input must be directly connected to a variable, and cannot be a constant or complex expression.</p> <p>This function is particularly designed for ST language. It allows simplified writing as assigning the result of the function is not mandatory.</p>
ST Language	<pre>IN := 1; Q := INC (IN); (* now: IN = 2 ; Q = 2 *) INC (IN); (* simplified call *)</pre>
FBD Language	
LD Language	
IL Language	Not available



2.9.13. Jumps JMP JMPC JMPNC JMPCN

Statement Jump to a label

Remarks A jump to a label branches the execution of the program after the specified label. Labels and jumps cannot be used in structured ST language. In FBD language, a jump is represented by the >> symbol followed by the label name. The input of the >> symbol must be connected to a valid Boolean signal. The jump is performed only if the input is TRUE. In LD language, the >> symbol, followed by the target label name, is used as a coil at the end of a rung. The jump is performed only if the rung state is TRUE. In IL language, JMP, JMPC, JMPCN and JMPNC instructions are used to specify a jump. The destination label is the operand of the jump instruction.



Attention

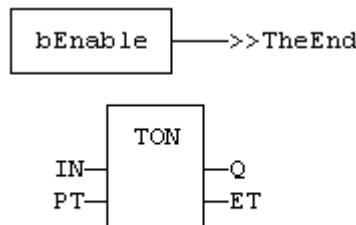
Backward jumps may lead to infinite loops that block the target cycle.

ST Language Not available

FBD Language In this Example the TON block will not be called if bEnable is *TRUE*:



Example

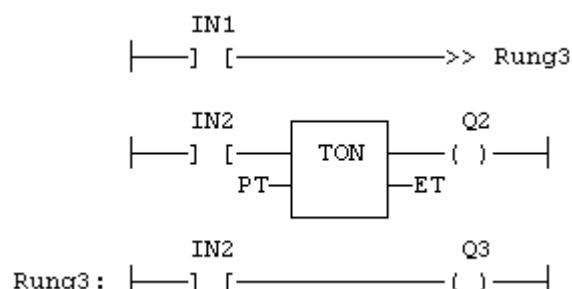


TheEnd:

LD Language In this Example the second rung will be skipped if IN1 is *TRUE*:



Example



IL Language Below is the meaning of possible jump instructions:

JMP Jump always

JMPC Jump if the current result is *TRUE*

JMPNC Jump if the current result is *FALSE*



JMPCN Same as JMPNC

Start: LD IN1

JMPC TheRest (* Jump to "TheRest" if IN1 is TRUE *)

LD IN2 (* these three instructions are not executed *)

ST Q2 (* if IN1 is TRUE *)

JMP TheEnd (* unconditional jump to "TheEnd" *)

TheRest: LD IN3

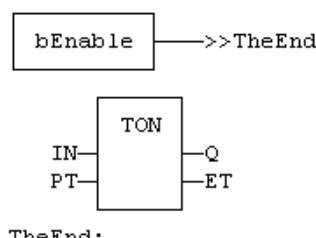
ST Q3

TheEnd:

2.9.14. Labels

Statement	Destination of a Jump instruction
Remarks	Labels are used as a destination of a jump instruction in FDB, LD or IL language. Labels and jumps cannot be used in structured ST language. A label must be represented by a unique name, followed by a colon (":"). In FBD language, labels can be inserted anywhere in the diagram, and are connected to nothing. In LD language, a label must identify a rung, and is shown on the left side of the rung. In IL language, labels are destination for JMP, JMPC, JMPCN and JMPNC instructions. They must be written before the instruction at the beginning of the line, and should index the beginning of a valid IL statement: LD (load) instruction, or unconditional instructions such as CAL, JMP or RET. The label can also be written alone on a line before the indexed instruction. In all languages, it is not mandatory that a label be a target of a jump instruction. You can also use label for marking parts of the programs in order to increase its readability.
ST Language	Not available
FBD Language	In this Example the TON block will not be called if bEnable is <i>TRUE</i> :

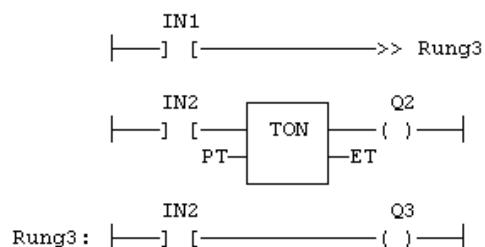
Example



TheEnd:

LD Language	In this Example the second rung will be skipped if IN1 is <i>TRUE</i> :

Example



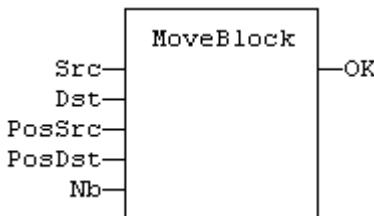
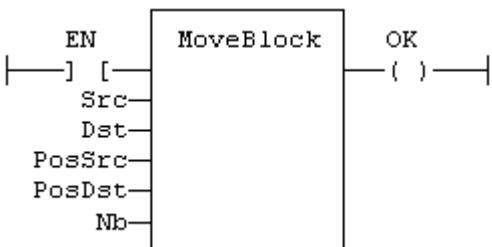
IL Language	Start: LD IN1 (* unused label - just for readability *) JMPC TheRest (* Jump to "TheRest" if IN1 is <i>TRUE</i> *)
	LD IN2 (* these two instructions are not executed *) ST Q2 (* if IN1 is <i>TRUE</i> *)



TheRest: LD IN3 (* label used as the jump destination *) ST Q3



2.9.15. MOVEBLOCK

Function	Move/Copy items of an array										
Inputs	<table><tr><td>SRC : ANY (*)</td><td>Array containing the source of the copy (*) SRC/DST cannot be a STRING.</td></tr><tr><td>DST : ANY (*)</td><td>Array containing the destination of the copy (*) SRC/DST cannot be a STRING.</td></tr><tr><td>PosSRC : DINT</td><td>Index of the first character in SRC</td></tr><tr><td>PosDST : DINT</td><td>Index of the destination in DST</td></tr><tr><td>NB : DINT</td><td>Number of items to be copied</td></tr></table>	SRC : ANY (*)	Array containing the source of the copy (*) SRC/DST cannot be a STRING.	DST : ANY (*)	Array containing the destination of the copy (*) SRC/DST cannot be a STRING.	PosSRC : DINT	Index of the first character in SRC	PosDST : DINT	Index of the destination in DST	NB : DINT	Number of items to be copied
SRC : ANY (*)	Array containing the source of the copy (*) SRC/DST cannot be a STRING.										
DST : ANY (*)	Array containing the destination of the copy (*) SRC/DST cannot be a STRING.										
PosSRC : DINT	Index of the first character in SRC										
PosDST : DINT	Index of the destination in DST										
NB : DINT	Number of items to be copied										
Outputs	OK : BOOL <i>TRUE</i> if successful										
Remarks	Arrays of string are not supported by this function. In LD language, the operation is executed only if the input rung (EN) is <i>TRUE</i> . The function is not available in IL language. The function copies NB consecutive items starting at the PosSRC index in SRC array to PosDST position in DST array. SRC and DST can be the same array. In that case, the function avoids lost items when source and destination areas overlap. This function checks array bounds and is always safe. The function returns TRUE if successful. It returns FALSE if input positions and number do not fit the bounds of SRC and DST arrays.										
ST Language	OK := MOVEBLOCK (SRC, DST, PosSRS, PosDST, NB);										
FBD Language											
LD Language	The function is executed only if EN is TRUE: 										
IL Language	Not available										



2.9.16. Parenthesis ()

Operator

Force the evaluation order in a complex expression

Remarks

Parenthesis are used in ST and IL language for changing the default evaluation order of various operations within a complex expression. For instance, the default evaluation of "2 * 3 + 4" expression in ST language gives a result of 10 as "*" operator has highest priority. Changing the expression as "2 * (3 + 4)" gives a result of 14. Parenthesis can be nested in a complex expression.

Below is the default evaluation order for ST language operations (first is highest priority):

Unary operators:	- NOT
Multiply/Divide:	* /
Add/Subtract:	+ -
Comparisons:	< > <= >= = <>
Boolean And:	& AND
Boolean Or:	OR
Exclusive OR:	XOR

In IL language, the default order is the sequence of instructions. Each new instruction modifies the current result sequentially. In IL language, the opening parenthesis "(" is written between the instruction and its operand. The closing parenthesis ")" must be written alone as an instruction without operand.

ST Language

`Q := (IN1 + (IN2 / IN 3)) * IN4;`

FBD Language

Not available

LD Language

Not available

IL Language

```
Op1: LD( IN1
      ADD( IN2
      MUL IN3
      )
      SUB IN4
      )
      ST Q (* Q is: (IN1 + (IN2 * IN3 ) - IN4 ) *)
```



See also Assignment



2.9.17. REPEAT UNTIL END_REPEAT

Statement Repeat a list of statements

Syntax

```
REPEAT
  <statements>
  UNTIL <BOOL expression> END_REPEAT;
```

Remarks The statements between REPEAT and UNTIL are executed until the Boolean expression is TRUE. The condition is evaluated after the statements are executed. Statements are executed at least once.

 **Attention**

Loop instructions may lead to infinite loops that block the target cycle. Never test the state of an input in the condition as the input will not be refreshed before the next cycle.

ST Language	<pre>iPos := 0; REPEAT MyArray[iPos] := 0; iNbCleared := iNbCleared + 1; iPos := iPos + 1; UNTIL iPos = iMax END_REPEAT;</pre>
FBD Language	Not available
LD Language	Not available
IL Language	Not available

2.9.18. RETURN RET RETC RETNC RETCN

Statement Jump to the end of the program

Remarks The *RETURN* statement jumps to the end of the program. In FBD language, the return statement is represented by the "<RETURN>" symbol. The input of the symbol must be connected to a valid Boolean signal. The jump is performed only if the input is *TRUE*. In LD language, the "<RETURN>" symbol is used as a coil at the end of a rung. The jump is performed only if the rung state is *TRUE*. In IL language, RET, RETC, RETCN and RETNC instructions are used.

When used within an action block of a SFC step, the *RETURN* statement jumps to the end of the action block.

ST Language

```
IF NOT bEnable THEN
    RETURN;
END_IF;
```

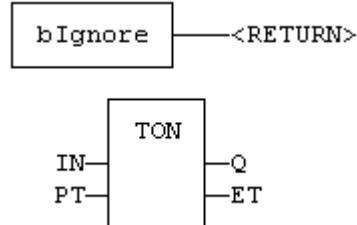
The rest of the program will not be executed if *bEnabled* is *FALSE*.

FBD Language



Example

In this Example the TON block will not be called if *bIgnore* is *TRUE*:

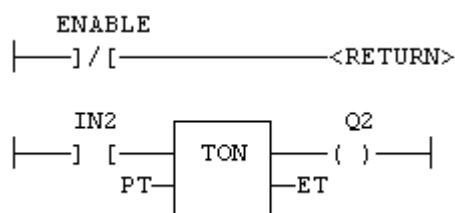


LD Language



Example

In this Example the second rung will be skipped if *ENABLE* is *FALSE*:



IL Language

Below is the meaning of possible instructions:

RET Jump to the end always.

RETC Jump to the end if the current result is *TRUE*.

RETNC Jump to the end if the current result is *FALSE*.

RETCN Same as RETNC.

Start: LD IN1

RETC (* Jump to the end if IN1 is TRUE *)

LD IN2 (* these instructions are not executed *)



ST Q2 (* if IN1 is TRUE *)
RET (* Jump to the end unconditionally *)

LD IN3 (* these instructions are never executed *)
ST Q3



2.9.19. WHILE DO END WHILE

Statement Repeat a list of statements

Syntax

```
WHILE <BOOL expression> DO
  <statements>
END WHILE ;
```

Remarks The statements between DO and END WHILE are executed while the boolean expression is **TRUE**. The condition is evaluated before the statements are executed. If the condition is **FALSE** when WHILE is first reached, statements are never executed.

 **Attention**

Loop instructions may lead to infinite loops that block the target cycle. Never test the state of an input in the condition as the input will not be refreshed before the next cycle.

ST Language	<pre>iPos := 0; WHILE iPos < iMax DO MyArray[iPos] := 0; iNbCleared := iNbCleared + 1; END WHILE;</pre>
FBD Language	Not available
LD Language	Not available
IL Language	Not available



2.9.20. ON

Statement Conditional execution of statements

Syntax

```
ON <BOOL expression> DO
  <statements>
END_DO;
```

Remarks

Statements within the ON structure are executed only when the boolean expression rises from FALSE to TRUE. The ON instruction avoids systematic use of the R_TRIG function block or other "last state" flags.

The ON syntax is available in any program, sub-program or UDFB. It is available in both T5 p-code or native code compilation modes.

This statement is an extension to the standard and is not IEC61131-3 compliant.

ST Language (* This Example counts the rising edges of variable bIN *)

```
ON bIN DO
  diCount := diCount + 1;
END_DO;
```



2.9.21. WAIT / WAIT_TIME

Statement Suspend the execution of a ST program

Syntax

```
WAIT <BOOL expression>;  
WAIT_TIME <TIME expression>;
```

Remarks The WAIT statement checks the attached boolean expression and:

- If the expression is TRUE, the program continues normally.
- If the expression is FALSE, then the execution of the program is suspended until the next PLC cycle. The Boolean expression is checked again during the next cycles until it becomes TRUE. The execution of other programs is not affected.

The WAIT_TIME statement suspends the execution of the program for the specified duration. The execution of other programs is not affected.

These instructions are available in ST language only and have no corresponding equivalent in other languages. These instructions cannot be called in a User Defined Function Block (UDFB).

The use of WAIT or WAIT_TIME in a UDFB provokes a compile error.

WAIT and WAIT_TIME instructions can be called in a sub-program. However, this may lead to some unsafe situation if the same sub program is called from various programs. Re-entrancy is not supported by WAIT and WAIT_TIME instructions. Avoiding this situation is the responsibility of the programmer. The compiler outputs some warning messages if a sub-program containing a WAIT or WAIT_TIME instruction is called from more than one program.

These instructions should not be called from ST parts of SFC programs. This makes no sense as SFC is already a state machine. The use of WAIT or WAIT_TIME in SFC or in a sub-program called from SFC provokes a compile error.

These instructions are not available when the code is compiled through a "C" compiler. Using "C" code generation with a program containing a WAIT or WAIT_TIME instruction provokes an error during post-compiling.

These statement are extensions to the standard and are not IEC61131-3 compliant.

ST Language (* use of WAIT with different kinds of BOOL expressions *)

```
WAIT BoolVariable;  
WAIT (diLevel > 100 ) AND NOT bAlarm;  
WAIT SubProgCall ();
```

(* use of WAIT_TIME with different kinds of TIME expressions *)

```
WAIT_TIME t#2s;  
WAIT_TIME TimeVariable;
```



2.10. Boolean Operations

Below are the standard operators for managing Booleans:

Operator	Description
AND	performs a Boolean AND
OR	performs a Boolean OR
XOR	performs an exclusive OR
NOT	performs a Boolean negation of its input
S	force a Boolean output to <i>TRUE</i>
R	force a Boolean output to <i>FALSE</i>

Below are the available blocks for managing Boolean signals:

Block	Description
RS	reset dominant bistable
SR	set dominant bistable
R_	rising pulse detection
F_TRIG	falling pulse detection
SEMA	semaphore
FLIPFLOP	flipflop^bistable

2.10.1. FLIPFLOP

Function Block Flipflop bistable

Inputs IN : BOOL Swap command (on rising edge).

RST : BOOL Reset to *FALSE*.

Outputs Q : BOOL Output

Remarks The output is systematically reset to *FALSE* if RST is *TRUE*.

The output changes on each rising edge of the IN input, if RST is *FALSE*.

ST Language MyFlipFlop is declared as an instance of FLIPFLOP function block:

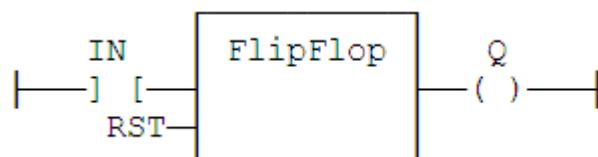
```
MyFlipFlop (IN, RST);
```

```
Q := MyFlipFlop.Q;
```

FBD Language



LD Language The IN command is the rung - the rung is the output:



IL Language MyFlipFlop is declared as an instance of FLIPFLOP function block:

```
Op1: CAL MyFlipFlop (IN, RST)
```

```
LD MyFlipFlop.Q
```

```
ST Q1
```

2.10.2. F_TRIG

Function Block Falling pulse detection

Inputs CLK : BOOL Boolean signal

Outputs Q : BOOL TRUE when the input changes from TRUE to FALSE

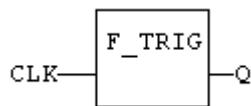
Truth table	CLK	CLK prev	Q
	0	0	0
	0	1	1
	1	0	0
	1	1	0

Remarks Although]P[an]N[contacts may be used in LD language, it is recommended to use declared instances of R_TRIG or F_TRIG function blocks in order to avoid unexpected behaviour during an On Line change.

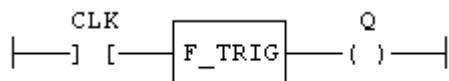
ST Language MyTrigger is declared as an instance of F_TRIG function block:

```
MyTrigger (CLK);  
Q := MyTrigger.Q;
```

FBD Language



LD Language The input signal is the rung, and the rung is also the output:



IL Language MyTrigger is declared as an instance of F_TRIG function block:

```
Op1: CAL MyTrigger (CLK)  
LD MyTrigger.Q  
ST Q
```



See also R_TRIG

2.10.3. AND ANDN &

Operator Performs a logical AND of all inputs

Inputs IN1 : BOOL First Boolean input
IN2 : BOOL Second Boolean input

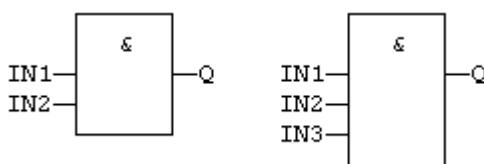
Outputs Q : BOOL Boolean AND of all inputs

Truth table	IN1	IN2	Q
	0	0	0
	0	1	0
	1	0	0
	1	1	1

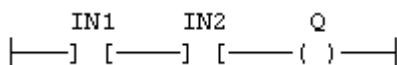
Remarks In FBD language, the block may have up to 16 inputs. The block is called "&" in FBD language. In LD language, an AND operation is represented by serialized contacts. In IL language, the AND instruction performs a logical AND between the current result and the operand. The current result must be Boolean. The ANDN instruction performs an AND between the current result and the Boolean negation of the operand. In ST and IL languages, "&" can be used instead of "AND".

ST Language `Q := IN1 AND IN2;`
`Q := IN1 & IN2 & IN3;`

FBD Language The block may have up to 16 inputs:



LD Language Serialized contacts:



IL Language `Op1: LD IN1`
`& IN2 (* "&" or "AND" can be used *)`
`ST Q (* Q is equal to: IN1 AND IN2 *)`
`Op2: LD IN1`
`AND IN2`
`&N IN3 (* "&N" or "ANDN" can be used *)`
`ST Q (* Q is equal to: IN1 AND IN2 AND (NOT IN3) *)`



2.10.4. NOT

Operator Performs a Boolean negation of the input

Inputs IN : BOOL Boolean value

Outputs Q : BOOL Boolean negation of the input

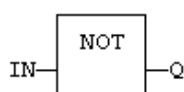
Truth table	IN	Q
	0	1
	1	0

Remarks In FBD language, the block *NOT* can be used. Alternatively, you can use a link terminated by a *o* negation. In LD language, negated contacts and coils can be used. In IL language, the *N* modifier can be used with instructions LD, AND, OR, XOR and ST. It represents a negation of the operand. In ST language, *NOT* can be followed by a complex Boolean expression between parenthesis.

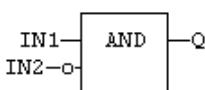
ST Language `Q := NOT IN;`

`Q := NOT (IN1 OR IN2);`

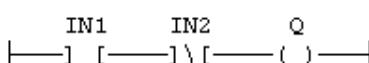
FBD Language Explicit use of the *NOT* block:



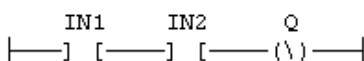
Use of a negated link: Q is IN1 AND NOT IN2:



LD Language Negated contact: Q is: IN1 AND NOT IN2:



Negated coil: Q is NOT (IN1 AND IN2):



IL Language Op1: LDN IN1

OR IN2

ST Q (* Q is equal to: (NOT IN1) OR IN2 *)

Op2: LD IN1

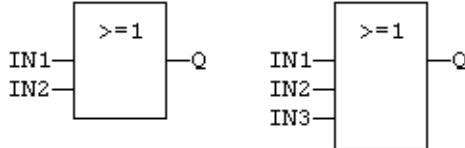
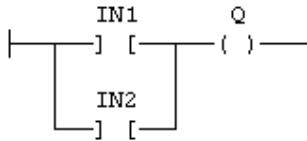
AND IN2

STN Q (* Q is equal to: NOT (IN1 AND IN2) *)

See also

AND OR XOR

2.10.5. OR ORN

Operator	Performs a logical OR of all inputs		
Inputs	IN1 : BOOL	First Boolean input	
	IN2 : BOOL	Second Boolean input	
Outputs	Q : BOOL Boolean OR of all inputs		
Truth table	IN1	IN2	Q
	0	0	0
	0	1	1
	1	0	1
	1	1	1
Remarks	In FBD language, the block may have up to 16 inputs. The block is called ≥ 1 in FBD language. In LD language, an <i>OR</i> operation is represented by contacts in parallel. In IL language, the <i>OR</i> instruction performs a logical <i>OR</i> between the current result and the operand. The current result must be Boolean. The <i>ORN</i> instruction performs an <i>OR</i> between the current result and the Boolean negation of the operand.		
ST Language	Q := IN1 OR IN2; Q := IN1 OR IN2 OR IN3;		
FBD Language	The block may have up to 16 inputs:		
			
LD Language	Parallel contacts:		
			
IL Language	Op1: LD IN1 OR IN2 ST Q (* Q is equal to: IN1 OR IN2 *) Op2: LD IN1 ORN IN2 ST Q (* Q is equal to: IN1 OR (NOT IN2) *)		



See also

AND XOR NOT



2.10.6. R

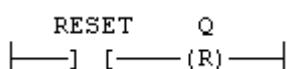
Operator	Force a Boolean output to <i>FALSE</i>		
Inputs	RESET : BOOL Condition		
Outputs	Q : BOOL Output to be forced		
Truth table	RESET	Q prev	Q
	0	0	0
	0	1	1
	1	0	0
	1	1	0

Remarks S and R operators are available as standard instructions in the IL language. In LD languages they are represented by (S) and (R) coils. In FBD language, you can use (S) and (R) coils, but you should prefer RS and SR function blocks. Set and reset operations are not available in ST language.

ST Language Not available

FBD Language Not available. Use RS or SR function blocks

LD Language Use of "R" coil:



IL Language Op1: LD RESET

R Q (* Q is forced to FALSE if RESET is TRUE *)

(* Q is unchanged if RESET is FALSE *)



See also

S

RS

SR

2.10.7. RS

Function Block Reset dominant bistable

Inputs SET : BOOL Condition for forcing to *TRUE*
RESET1 : BOOL Condition for forcing to *FALSE* (highest priority command)

Outputs Q1 : BOOL Output to be forced

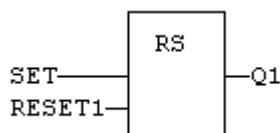
Truth table	SET	RESET1	Q1 prev	Q1
	0	0	0	0
	0	0	1	1
	0	1	0	0
	0	1	1	0
	1	0	0	1
	1	0	1	1
	1	1	0	0
	1	1	1	0

Remarks The output is unchanged when both inputs are *FALSE*. When both inputs are *TRUE*, the output is forced to *FALSE* (reset dominant).

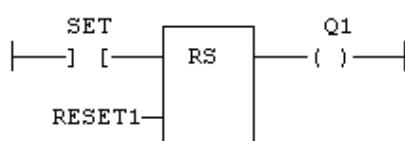
ST Language MyRS is declared as an instance of RS function block:

```
MyRS (SET, RESET1 );  
Q1 := MyRS.Q1;
```

FBD Language



LD Language The SET command is the rung - the rung is the output:



IL Language MyRS is declared as an instance of RS function block:

```
Op1: CAL MyRS (SET, RESET1 )  
LD MyRS.Q1  
ST Q1
```



See also

R

S

SR



2.10.8. R_TRIG

Function Block Rising pulse detection

Inputs CLK : BOOL Boolean signal

Outputs Q : BOOL TRUE when the input changes from FALSE to TRUE

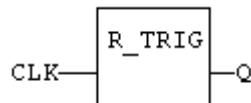
Truth table	CLK	CLK prev	Q
	0	0	0
	0	1	0
	1	0	1
	1	1	0

Remarks Although]P[an]N[contacts may be used in LD language, it is recommended to use declared instances of R_TRIG or F_TRIG function blocks in order to avoid unexpected behaviour during an On Line change.

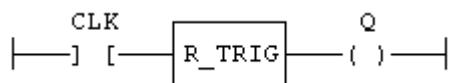
ST Language MyTrigger is declared as an instance of R_TRIG function block:

```
MyTrigger (CLK);  
Q := MyTrigger.Q;
```

FBD Language



LD Language The input signal is the rung - the rung is the output:



IL Language MyTrigger is declared as an instance of R_TRIG function block:

```
Op1: CAL MyTrigger (CLK)  
LD MyTrigger.Q  
ST Q
```



See also

[F_TRIG](#)



2.10.9. S

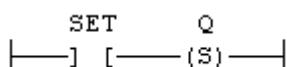
Operator	Force a Boolean output to <i>TRUE</i>		
Inputs	SET : BOOL Condition		
Outputs	Q : BOOL Output to be forced		
Truth table	SET	Q prev	Q
	0	0	0
	0	1	1
	1	0	1
	1	1	1

Remarks S and R operators are available as standard instructions in the IL language. In LD languages they are represented by (S) and (R) coils. In FBD language, you can use (S) and (R) coils, but you should prefer RS and SR function blocks. Set and reset operations are not available in ST language.

ST Language Not available

FBD Language Not available. Use RS or SR function blocks

LD Language Use of S coil:



IL Language Op1: LD SET

S Q (* Q is forced to TRUE if SET is TRUE *)
(* Q is unchanged if SET is FALSE *)



See also

R

RS

SR

2.10.10. SEMA

Function Block Semaphore

Inputs CLAIM : BOOL Takes the semaphore
 RELEASE : BOOL Releases the semaphore

Outputs BUSY : BOOL *TRUE* if semaphore is busy

Remarks The function block implements the following algorithm:

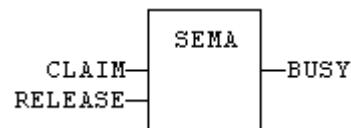
```
BUSY := mem;  
if CLAIM then  
    mem := TRUE;  
else if RELEASE then  
    BUSY := FALSE;  
    mem := FALSE;  
end_if;
```

In LD language, the input rung is the CLAIM command. The output rung is the BUSY output signal.

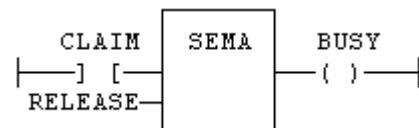
ST Language MySema is a declared instance of SEMA function block:

```
MySema (CLAIM, RELEASE );  
BUSY := MyBlinker.BUSY;
```

FBD Language



LD Language



IL Language

MySema is a declared instance of SEMA function block:

```
Op1: CAL MySema (CLAIM, RELEASE )  
LD MyBlinker.BUSY  
ST BUSY
```



2.10.11. SR

Function Block Set dominant bistable

Inputs SET1 : BOOL Condition for forcing to TRUE (highest priority command).
RESET : BOOL Condition for forcing to FALSE.

Outputs Q1 : BOOL Output to be forced

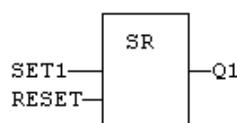
Truth table	SET1	RESET	Q1 prev	Q1
0	0	0	0	0
0	0	1	1	1
0	1	0	0	0
0	1	1	1	0
1	0	0	0	1
1	0	1	1	1
1	1	0	1	1
1	1	1	1	1

Remarks The output is unchanged when both inputs are FALSE. When both inputs are TRUE, the output is forced to TRUE (set dominant).

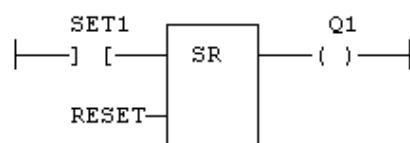
ST Language MySR is declared as an instance of SR function block:

```
MySR (SET1, RESET );  
Q1 := MySR.Q1;
```

FBD Language



LD Language The SET1 command is the rung - the rung is the output:



IL Language MySR is declared as an instance of SR function block:

```
Op1: CAL MySR (SET1, RESET )  
LD MySR.Q1  
ST Q1
```



See also

R

S

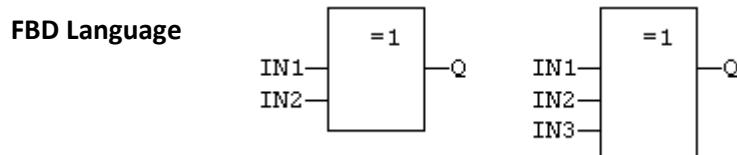
RS

2.10.12. XOR XORN

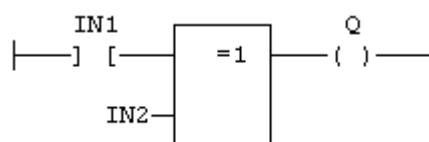
Operator	Performs an exclusive <i>OR</i> of all inputs		
Inputs	IN1 : BOOL	First Boolean input	
	IN2 : BOOL	Second Boolean input	
Outputs	Q : BOOL Exclusive <i>OR</i> of all inputs		
Truth table	IN1	IN2	Q
	0	0	0
	0	1	1
	1	0	1
	1	1	0

Remarks The block is called `=1` in FBD and LD languages. In IL language, the *XOR* instruction performs an exclusive *OR* between the current result and the operand. The current result must be Boolean. The *XORN* instruction performs an exclusive between the current result and the Boolean negation of the operand.

ST Language `Q := IN1 XOR IN2;`
`Q := IN1 XOR IN2 XOR IN3;`



LD Language First input is the rung. The rung is the output:



IL Language

`Op1: LD IN1`
`XOR IN2`
`ST Q (* Q is equal to: IN1 XOR IN2 *)`

`Op2: LD IN1`
`XORN IN2`
`ST Q (* Q is equal to: IN1 XOR (NOT IN2) *)`



See also

AND OR NOT



2.11. Arithmetic operations

Below are the standard operators that perform arithmetic operations:

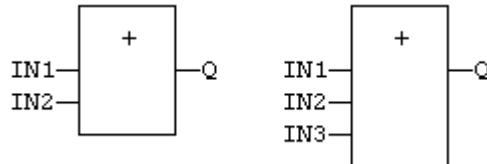
Operator	Description
+	Addition
-	subtraction
*	multiplication
/	division
- (NEG)	integer negation (unary operator)

Below are the standard functions that perform arithmetic operations:

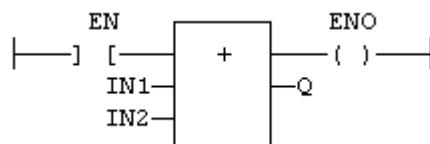
Function	Description
MIN	get the minimum of two values
MAX	get the maximum of two values
LIMIT	bound an integer to low and high limits
MOD	modulo
ODD	test if an integer is odd
SetWithin	force a value when inside an interval

2.11.1. + ADD

Operator	Performs an addition of all inputs
Inputs	IN1 : ANY First input IN2 : ANY Second input
Outputs	Q : ANY Result: IN1 + IN2
Remarks	All inputs and the output must have the same type. In FBD language, the block may have up to 16 inputs. In LD language, the input rung (EN) enables the operation, and the output rung keeps the same value as the input rung. In IL language, the ADD instruction performs an addition between the current result and the operand. The current result and the operand must have the same type. The addition can be used with strings. The result is the concatenation of the input strings.
ST Language	<code>Q := IN1 + IN2;</code> <code>MyString := 'He' + 'll' + 'o'; (* MyString is equal to 'Hello' *)</code>
FBD Language	The block may have up to 16 inputs:



LD Language	The addition is executed only if EN is <i>TRUE</i> . ENO is equal to EN.
--------------------	---



IL Language	Op1: LD IN1 ADD IN2 ST Q (* Q is equal to: IN1 + IN2 *) Op2: LD IN1 ADD IN2 ADD IN3 ST Q (* Q is equal to: IN1 + IN2 + IN3 *)
--------------------	---

2.11.2. / DIV

Operator	Performs a division of inputs
Inputs	IN1 : ANY_NUM First input IN2 : ANY_NUM Second input
Outputs	Q : ANY_NUM Result: IN1 / IN2
Remarks	All inputs and the output must have the same type. In LD language, the input rung (EN) enables the operation, and the output rung keeps the same value as the input rung. In IL language, the DIV instruction performs a division between the current result and the operand. The current result and the operand must have the same type.
ST Language	<code>Q := IN1 / IN2;</code>
FBD Language	
LD Language	The division is executed only if EN is <i>TRUE</i> . ENO is equal to EN.
IL Language	<code>Op1: LD IN1 DIV IN2 ST Q (* Q is equal to: IN1 / IN2 *)</code> <code>Op2: LD IN1 DIV IN2 DIV IN3 ST Q (* Q is equal to: IN1 / IN2 / IN3 *)</code>

2.11.3. NEG -

Operator Performs an integer negation of the input

Inputs IN : DINT Integer value

Outputs Q : DINT Integer negation of the input

**Truth table
(Examples)** IN Q

0 0

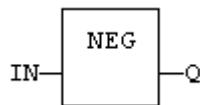
1 -1

-123 123

Remarks In FBD and LD language, the block NEG can be used. In LD language, the operation is executed only if the input rung (EN) is *TRUE*. The output rung (ENO) keeps the same value as the input rung. This feature is not available in IL language. In ST language, "-" can be followed by a complex Boolean expression between parenthesis.

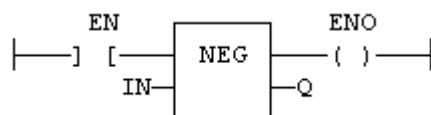
ST Language $Q := -IN;$
 $Q := - (IN1 + IN2);$

FBD Language



LD Language The negation is executed only if EN is *TRUE*.

ENO keeps the same value as EN.



IL Language Not available

2.11.4. LIMIT

Operator Bounds an integer between low and high limits

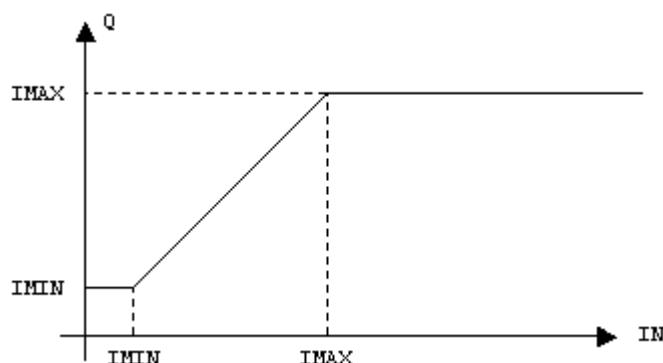
Inputs IMIN : DINT Low bound

IN : DINT Input value

IMAX : DINT High bound

Outputs Q : DINT IMIN if IN < IMIN; IMAX if IN > IMAX; IN otherwise

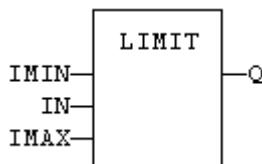
Function diagram



Remarks In LD language, the input rung (EN) enables the operation, and the output rung keeps the state of the input rung. In IL language, the first input must be loaded before the function call. Other inputs are operands of the function, separated by a coma.

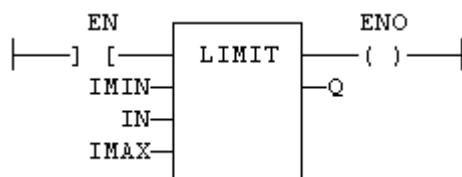
ST Language `Q := LIMIT (IMIN, IN, IMAX);`

FBD Language



LD Language The comparison is executed only if EN is *TRUE*.

ENO has the same value as EN.



IL Language

```
Op1: LD  IMIN
      LIMIT IN, IMAX
      ST  Q
```

2.11.5. MAX

Operator Get the maximum of two values

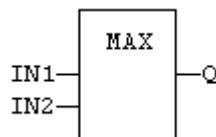
Inputs IN1 : ANY First input
IN2 : ANY Second input

Outputs Q : ANY IN1 if IN1 > IN2; IN2 otherwise

Remarks In LD language, the input rung (EN) enables the operation, and the output rung keeps the state of the input rung. In IL language, the first input must be loaded before the function call. The second input is the operand of the function.

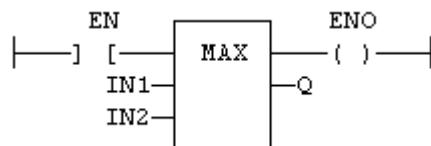
ST Language `Q := MAX (IN1, IN2);`

FBD Language



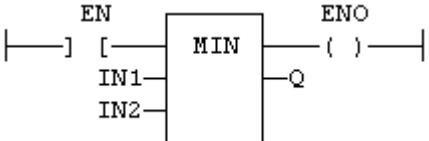
LD Language The comparison is executed only if EN is *TRUE*.

ENO has the same value as EN.



IL Language
`Op1: LD IN1
MAX IN2
ST Q (* Q is the maximum of IN1 and IN2 *)`

2.11.6. MIN

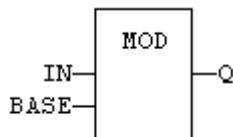
Operator	Get the minimum of two values
Inputs	IN1 : ANY First input IN2 : ANY Second input
Outputs	Q : ANY IN1 if IN1 < IN2; IN2 otherwise
Remarks	In LD language, the input rung (EN) enables the operation, and the output rung keeps the state of the input rung. In IL language, the first input must be loaded before the function call. The second input is the operand of the function.
ST Language	<code>Q := MIN (IN1, IN2);</code>
FBD Language	
LD Language	The comparison is executed only if EN is <i>TRUE</i> . ENO has the same value as EN. 
L Language	<code>Op1: LD IN1 MIN IN2 ST Q (* Q is the minimum of IN1 and IN2 *)</code>

2.11.7. MOD / MODR / MODLR

Operator	Calculation of modulo	
Inputs	IN : DINT/REAL/LREAL	Input value
	BASE : DINT/REAL/LREAL	Base of the modulo
Outputs	Q : DINT/REAL/LREAL	Modulo: rest of the integer division (IN / BASE)
Remarks	In LD language, the input rung (EN) enables the operation, and the output rung keeps the state of the input rung. In IL language, the first input must be loaded before the function call. The second input is the operand of the function.	

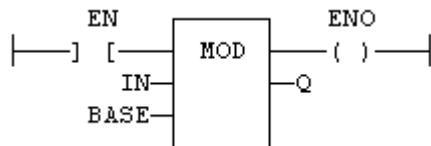
ST Language `Q := MOD (IN, BASE);`

FBD Language



LD Language The comparison is executed only if EN is *TRUE*.

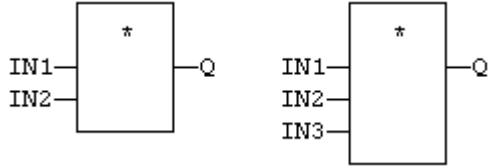
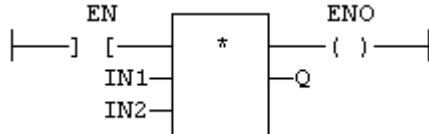
ENO has the same value as EN.



IL Language

Op1: LD IN
MOD BASE
ST Q (* Q is the rest of integer division: IN / BASE *)

2.11.8. * MUL

Operator	Performs a multiplication of all inputs.
Inputs	IN1 : ANY_NUM First input. IN2 : ANY_NUM Second input.
Outputs	Q : ANY_NUM Result: IN1 * IN2.
Remarks	All inputs and the output must have the same type. In FBD language, the block may have up to 16 inputs. In LD language, the input rung (EN) enables the operation, and the output rung keeps the same value as the input rung. In IL language, the MUL instruction performs a multiplication between the current result and the operand. The current result and the operand must have the same type.
ST Language	<code>Q := IN1 * IN2;</code>
FBD Language	The block may have up to 16 inputs: 
LD Language	The multiplication is executed only if EN is <i>TRUE</i> . ENO is equal to EN. 
IL Language	<code>Op1: LD IN1 MUL IN2 ST Q (* Q is equal to: IN1 * IN2 *)</code> <code>Op2: LD IN1 MUL IN2 MUL IN3 ST Q (* Q is equal to: IN1 * IN2 * IN3 *)</code>

2.11.9. ODD

Operator Test if an integer is odd

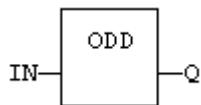
Inputs IN : DINT Input value

Outputs Q : BOOL *TRUE* if IN is odd. *FALSE* if IN is even.

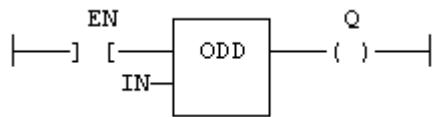
Remarks In LD language, the input rung (EN) enables the operation, and the output rung is the result of the function. In IL language, the input must be loaded before the function call.

ST Language `Q := ODD (IN);`

FBD Language



LD Language



IL Language

Op1: LD IN
ODD
ST Q (* Q is TRUE if IN is odd *)

2.11.10. - SUB

Operator	Performs a subtraction of inputs	
Inputs	IN1 : ANY_NUM / TIME	First input
	IN2 : ANY_NUM / TIME	Second input
Outputs	Q : ANY_NUM / TIME	Result: IN1 - IN2
Remarks	All inputs and the output must have the same type. In LD language, the input rung (EN) enables the operation, and the output rung keeps the same value as the input rung. In IL language, the SUB instruction performs a subtraction between the current result and the operand. The current result and the operand must have the same type.	
ST Language	<code>Q := IN1 - IN2;</code>	
FBD Language		
LD Language	The subtraction is executed only if EN is <i>TRUE</i> . ENO is equal to EN.	
IL Language	Op1: LD IN1 SUB IN2 ST Q (* Q is equal to: IN1 - IN2 *) Op2: LD IN1 SUB IN2 SUB IN3 ST Q (* Q is equal to: IN1 - IN2 - IN3 *)	



2.11.11. SetWithin

Function Force a value when inside an interval

Inputs

IN : ANY	Input
MIN : ANY	Low limit of the interval
MAX : ANY	High limit of the interval
VAL : ANY	Value to apply when inside the interval

Outputs Q : BOOL Result

Truth table

IN	Q
IN < MIN	IN
IN > MAX	IN
MIN < IN < MAX	VAL

Remarks

The output is forced to VAL when the IN value is within the [MIN .. MAX] interval. It is set to IN when outside the interval.



2.12. Comparison operations

Below are the standard operators and blocks that perform comparisons:

Operator	Meaning
<	less than
>	greater than
<=	less or equal
>=	greater or equal
=	is equal
<>	is not equal
CMP	detailed comparison

2.12.1. CMP

Function Block Comparison with detailed outputs for integer inputs

Inputs IN1 : DINT First value

IN2 : DINT Second value

Outputs LT : BOOL *TRUE* if IN1 < IN2

EQ : BOOL *TRUE* if IN1 = IN2

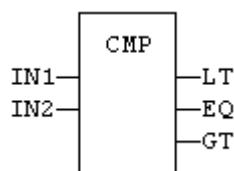
GT : BOOL *TRUE* if IN1 > IN2

Remarks In LD language, the rung input (EN) validates the operation. The rung output is the result of LT (lower than comparison).

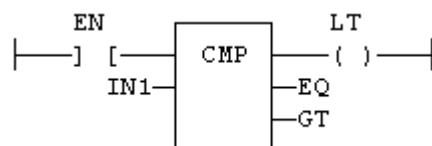
ST Language MyCmp is declared as an instance of CMP function block:

```
MyCMP (IN1, IN2);  
bLT := MyCmp.LT;  
bEQ := MyCmp.EQ;  
bGT := MyCmp.GT;
```

FBD Language



LD Language The comparison is performed only if EN is *TRUE*:



IL Language MyCmp is declared as an instance of CMP function block:

```
Op1: CAL MyCmp (IN1, IN2)  
LD MyCmp.LT  
ST bLT  
LD MyCmp.EQ  
ST bEQ  
LD MyCmp.GT  
ST bGT
```



2.12.2. \geq GE

Function *Operator - Test if first input is greater than or equal to second input*

Inputs IN1 : ANY First input

IN2 : ANY Second input

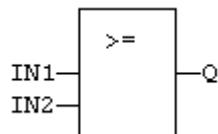
Outputs Q : BOOL *TRUE if IN1 \geq IN2*

Remarks Both inputs must have the same type. In LD language, the input rung (EN) enables the operation, and the output rung is the result of the comparison. In IL language, the GE instruction performs the comparison between the current result and the operand. The current result and the operand must have the same type.

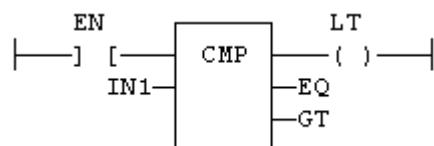
Comparisons can be used with strings. In that case, the lexical order is used for comparing the input strings. For instance, "ABC" is less than "ZX" ; "ABCD" is greater than "ABC".

ST Language `Q := IN1 \geq IN2;`

FBD Language



LD Language The comparison is executed only if EN is *TRUE*:



IL Language `Op1: LD IN1`

`GE IN2`

`ST Q (* Q is true if IN1 \geq IN2 *)`

2.12.3. > GT

Function	<i>Operator</i> - Test if first input is greater than second input
Inputs	IN1 : ANY First input IN2 : ANY Second input
Outputs	Q : BOOL <i>TRUE</i> if IN1 > IN2
Remarks	<p>Both inputs must have the same type. In LD language, the input rung (EN) enables the operation, and the output rung is the result of the comparison. In IL language, the GT instruction performs the comparison between the current result and the operand. The current result and the operand must have the same type.</p> <p>Comparisons can be used with strings. In that case, the lexical order is used for comparing the input strings. For instance, "ABC" is less than "ZX" ; "ABCD" is greater than "ABC".</p>
ST Language	<code>Q := IN1 > IN2;</code>
FBD Language	
LD Language	The comparison is executed only if EN is <i>TRUE</i> :
IL Language	<code>Op1: LD IN1 GT IN2 ST Q (* Q is true if IN1 > IN2 *)</code>



2.12.4. = EQ

Function *Operator - Test if first input is equal to second input*

Inputs IN1 : ANY First input

IN2 : ANY Second input

Outputs Q : BOOL *TRUE if IN1 = IN2*

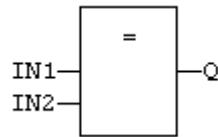
Remarks Both inputs must have the same type. In LD language, the input rung (EN) enables the operation, and the output rung is the result of the comparison. In IL language, the EQ instruction performs the comparison between the current result and the operand. The current result and the operand must have the same type.

Comparisons can be used with strings. In that case, the lexical order is used for comparing the input strings. For instance, "ABC" is less than "ZX" ; "ABCD" is greater than "ABC".

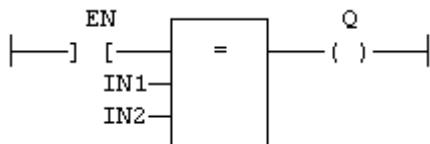
Equality comparisons cannot be used with TIME variables. The reason why is that the timer actually has the resolution of the target cycle and test may be unsafe as some values may never be reached.

ST Language `Q := IN1 = IN2;`

FBD Language



LD Language The comparison is executed only if EN is *TRUE*:



IL Language `Op1: LD IN1`

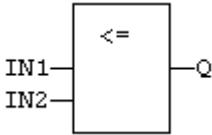
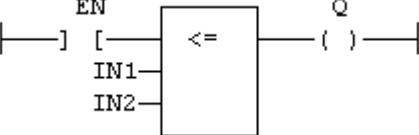
`EQ IN2`

`ST Q (* Q is true if IN1 = IN2 *)`

2.12.5. <> NE

Function	<i>Operator</i> - Test if first input is not equal to second input
Inputs	IN1 : ANY First input. IN2 : ANY Second input
Outputs	Q : BOOL <i>TRUE</i> if IN1 is not equal to IN2
Remarks	<p>Both inputs must have the same type. In LD language, the input rung (EN) enables the operation, and the output rung is the result of the comparison. In IL language, the NE instruction performs the comparison between the current result and the operand. The current result and the operand must have the same type.</p> <p>Comparisons can be used with strings. In that case, the lexical order is used for comparing the input strings. For instance, "ABC" is less than "ZX" ; "ABCD" is greater than "ABC".</p> <p>Equality comparisons cannot be used with TIME variables. The reason why is that the timer actually has the resolution of the target cycle and test may be unsafe as some values may never be reached.</p>
ST Language	<code>Q := IN1 <> IN2;</code>
FBD Language	
LD Language	The comparison is executed only if EN is <i>TRUE</i> :
IL Language	<code>Op1: LD IN1 NE IN2 ST Q (* Q is true if IN1 is not equal to IN2 *)</code>

2.12.6. <= LE

Function	<i>Operator</i> - Test if first input is less than or equal to second input
Inputs	IN1 : ANY First input IN2 : ANY Second input
Outputs	Q : BOOL <i>TRUE</i> if IN1 <= IN2
Remarks	<p>Both inputs must have the same type. In LD language, the input rung (EN) enables the operation, and the output rung is the result of the comparison. In IL language, the LE instruction performs the comparison between the current result and the operand. The current result and the operand must have the same type.</p> <p>Comparisons can be used with strings. In that case, the lexical order is used for comparing the input strings. For instance, "ABC" is less than "ZX" ; "ABCD" is greater than "ABC".</p>
ST Language	<code>Q := IN1 <= IN2;</code>
FBD Language	
LD Language	The comparison is executed only if EN is <i>TRUE</i> : 
IL Language	<code>Op1: LD IN1 LE IN2 ST Q (* Q is true if IN1 <= IN2 *)</code>

2.12.7. < LT

Function	<i>Operator</i> - Test if first input is less than second input
Inputs	IN1 : ANY First input IN2 : ANY Second input
Outputs	Q : BOOL <i>TRUE</i> if IN1 < IN2
Remarks	<p>Both inputs must have the same type. In LD language, the input rung (EN) enables the operation, and the output rung is the result of the comparison. In IL language, the LT instruction performs the comparison between the current result and the operand. The current result and the operand must have the same type.</p> <p>Comparisons can be used with strings. In that case, the lexical order is used for comparing the input strings. For instance, "ABC" is less than "ZX" ; "ABCD" is greater than "ABC".</p>
ST Language	<code>Q := IN1 < IN2;</code>
FBD Language	
LD Language	The comparison is executed only if EN is <i>TRUE</i> :
IL Language	<code>Op1: LD IN1 LT IN2 ST Q (* Q is true if IN1 < IN2 *)</code>



2.13. Type conversion functions

Below are the standard functions for converting a data into another data type:

Function	Conversion
ANY_TO_BOOL	converts to Boolean
ANY_TO_SINT	converts to small (8 bit) integer
ANY_TO_INT	converts to 16 bit integer
ANY_TO_DINT	converts to integer (32 bit - default)
ANY_TO_LINT	converts to long (64 bit) integer
ANY_TO_REAL	converts to real
ANY_TO_LREAL	converts to double precision real
ANY_TO_TIME	converts to time
ANY_TO_STRING	converts to character string
NUM_TO_STRING	converts a number to a string

Below are the standard functions performing conversions in BCD format. However, it should be noted that the BCD conversion functions may not be supported by all targets:

Function	Conversion
BIN_TO_BCD	converts a binary value to a DCB value
BCD_TO_BIN	converts a BCD value to a binary value



2.13.1. ANY_TO_BOOL

Function *Operator* - Converts the input into Boolean value

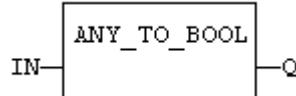
Inputs IN : ANY Input value

Outputs Q : BOOL Value converted to Boolean

Remarks For DINT, REAL and TIME input data types, the result is *FALSE* if the input is 0. The result is *TRUE* in all other cases. For STRING inputs, the output is *TRUE* if the input string is not empty, and *FALSE* if the string is empty. In LD language, the conversion is executed only if the input rung (EN) is *TRUE*. The output rung is the result of the conversion. In IL Language, the ANY_TO_BOOL function converts the current result.

ST Language `Q := ANY_TO_BOOL (IN);`

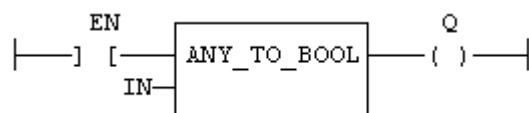
FBD Language



LD Language The conversion is executed only if EN is *TRUE*.

The output rung is the result of the conversion.

The output rung is *FALSE* if the EN is *FALSE*.



IL Language `Op1: LD IN`

`ANY_TO_BOOL`

`ST Q`



2.13.2. ANY_TO_DINT / ANY_TO_UDINT

Function *Operator* - Converts the input into integer value

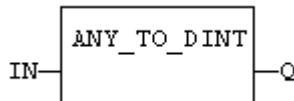
Inputs IN : ANY Input value

Outputs Q : DINT Value converted to integer

Remarks For BOOL input data types, the output is 0 or 1. For REAL input data type, the output is the integer part of the input real. For TIME input data types, the result is the number of milliseconds. For STRING inputs, the output is the number represented by the string, or 0 if the string does not represent a valid number. In LD language, the conversion is executed only if the input rung (EN) is TRUE. The output rung (ENO) keeps the same value as the input rung. In IL Language, the ANY_TO_DINT function converts the current result.

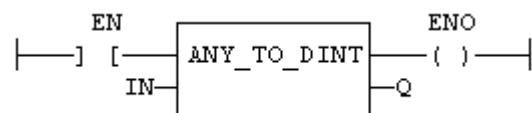
ST Language `Q := ANY_TO_DINT (IN);`

FBD Language



LD Language The conversion is executed only if EN is TRUE.

ENO keeps the same value as EN.



IL Language `Op1: LD IN`

`ANY_TO_DINT`

`ST Q`



2.13.3. ANY_TO_INT / ANY_TO_UINT

Function *Operator* - Converts the input into 16 bit integer value

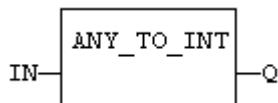
Inputs IN : ANY Input value

Outputs Q : INT Value converted to 16 bit integer

Remarks For BOOL input data types, the output is 0 or 1. For REAL input data type, the output is the integer part of the input real. For TIME input data types, the result is the number of milliseconds. For STRING inputs, the output is the number represented by the string, or 0 if the string does not represent a valid number. In LD language, the conversion is executed only if the input rung (EN) is TRUE. The output rung (ENO) keeps the same value as the input rung. In IL Language, the ANY_TO_INT function converts the current result.

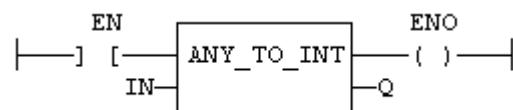
ST Language `Q := ANY_TO_INT (IN);`

FBD Language



LD Language The conversion is executed only if EN is TRUE.

ENO keeps the same value as EN.



IL Language `Op1: LD IN`

`ANY_TO_INT`

`ST Q`



2.13.4. ANY_TO_LINT

Function *Operator* - Converts the input into long (64 bit) integer value

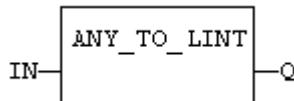
Inputs IN : ANY Input value

Outputs Q : LINT Value converted to long (64 bit) integer

Remarks For BOOL input data types, the output is 0 or 1. For REAL input data type, the output is the integer part of the input real. For TIME input data types, the result is the number of milliseconds. For STRING inputs, the output is the number represented by the string, or 0 if the string does not represent a valid number. In LD language, the conversion is executed only if the input rung (EN) is *TRUE*. The output rung (ENO) keeps the same value as the input rung. In IL Language, the ANY_TO_LINT function converts the current result.

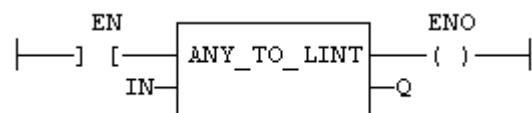
ST Language `Q := ANY_TO_LINT (IN);`

FBD Language



LD Language The conversion is executed only if EN is *TRUE*.

ENO keeps the same value as EN.



IL Language `Op1: LD IN`

`ANY_TO_LINT`

`ST Q`

2.13.5. ANY_TO_LREAL

Function *Operator* - Converts the input into double precision real value

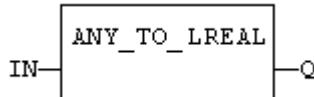
Inputs IN : ANY Input value

Outputs Q : LREAL Value converted to double precision real

Remarks For BOOL input data types, the output is 0.0 or 1.0. For DINT input data type, the output is the same number. For TIME input data types, the result is the number of milliseconds. For STRING inputs, the output is the number represented by the string, or 0.0 if the string does not represent a valid number. In LD language, the conversion is executed only if the input rung (EN) is *TRUE*. The output rung (ENO) keeps the same value as the input rung. In IL Language, the ANY_TO_LREAL function converts the current result.

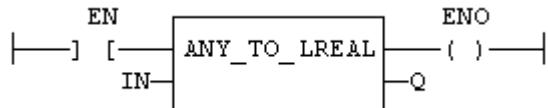
ST Language `Q := ANY_TO_LREAL (IN);`

FBD Language



LD Language The conversion is executed only if EN is *TRUE*.

ENO keeps the same value as EN.



IL Language Op1: LD IN

ANY_TO_LREAL

ST Q

2.13.6. ANY_TO_REAL

Function *Operator* - Converts the input into real value

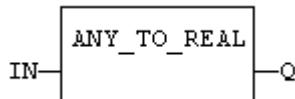
Inputs IN : ANY Input value

Outputs Q : REAL Value converted to real

Remarks For BOOL input data types, the output is 0.0 or 1.0. For DINT input data type, the output is the same number. For TIME input data types, the result is the number of milliseconds. For STRING inputs, the output is the number represented by the string, or 0.0 if the string does not represent a valid number. In LD language, the conversion is executed only if the input rung (EN) is *TRUE*. The output rung (ENO) keeps the same value as the input rung. In IL Language, the ANY_TO_REAL function converts the current result

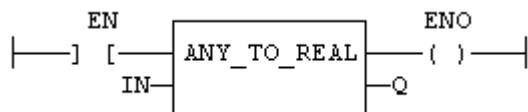
ST Language `Q := ANY_TO_REAL (IN);`

FBD Language



LD Language The conversion is executed only if EN is *TRUE*.

ENO keeps the same value as EN.



IL Language Op1: LD IN

ANY_TO_REAL

ST Q



2.13.7. ANY_TO_TIME

Function Operator - Converts the input into real value

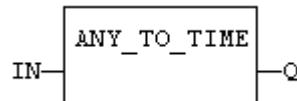
Inputs IN : ANY Input value

Outputs Q : TIME Value converted to time

Remarks For BOOL input data types, the output is t#0ms or t#1ms. For DINT or REAL input data type, the output is the time represented by the input number as a number of milliseconds. For STRING inputs, the output is the time represented by the string, or t#0ms if the string does not represent a valid time. In LD language, the conversion is executed only if the input rung (EN) is *TRUE*. The output rung (ENO) keeps the same value as the input rung. In IL Language, the ANY_TO_TIME function converts the current result.

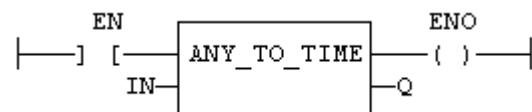
ST Language `Q := ANY_TO_TIME (IN);`

FBD Language



LD Language The conversion is executed only if EN is *TRUE*.

ENO keeps the same value as EN.



IL Language Op1: LD IN
 ANY_TO_TIME
 ST Q



2.13.8. ANY_TO_STRING

Function *Operator* - Converts the input into string value

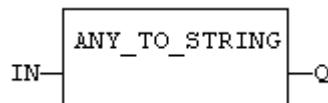
Inputs IN : ANY Input value

Outputs Q : STRING Value converted to string

Remarks For BOOL input data types, the output is 1 or 0 for *TRUE* and *FALSE* respectively. For DINT, REAL or TIME input data types, the output is the string representation of the input number. This is a number of milliseconds for TIME inputs. In LD language, the conversion is executed only if the input rung (EN) is *TRUE*. The output rung (ENO) keeps the same value as the input rung. In IL language, the ANY_TO_STRING function converts the current result.

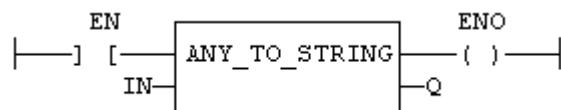
ST Language `Q := ANY_TO_STRING (IN);`

FBD Language



LD Language The conversion is executed only if EN is *TRUE*.

ENO keeps the same value as EN.



IL Language

Op1: LD IN

ANY_TO_STRING

ST Q



2.13.9. ANY_TO_SINT

Function *Operator* - Converts the input into a small (8 bit) integer value

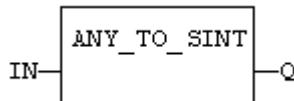
Inputs IN : ANY Input value

Outputs Q : SINT Value converted to a small (8 bit) integer

Remarks For BOOL input data types, the output is 0 or 1. For REAL input data type, the output is the integer part of the input real. For TIME input data types, the result is the number of milliseconds. For STRING inputs, the output is the number represented by the string, or 0 if the string does not represent a valid number. In LD language, the conversion is executed only if the input rung (EN) is *TRUE*. The output rung (ENO) keeps the same value as the input rung. In IL Language, the ANY_TO_SINT function converts the current result.

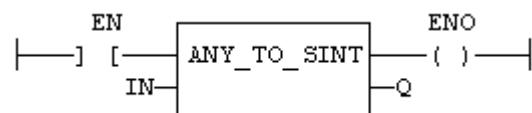
ST Language `Q := ANY_TO_SINT (IN);`

FBD Language



LD Language The conversion is executed only if EN is *TRUE*.

ENO keeps the same value as EN.



IL Language `Op1: LD IN`

`ANY_TO_SINT`

`ST Q`



2.13.10. NUM_TO_STRING

Function	Converts a number into string value	
Inputs	IN : ANY	Input number.
	WIDTH : DINT	Desired length for the output string (see remarks)
	DIGITS : DINT	Number of digits after decimal point
Outputs	Q : STRING	
Remarks	<p>This function converts any numerical value to a string. Unlike the ANY_TO_STRING function, it allows you to specify a desired length and a number of digits after the decimal points.</p> <p>If WIDTH is 0, the string is formatted with the necessary length.</p> <p>If WIDTH is greater than 0, the string is completed with leading blank characters in order to match the value of WIDTH.</p> <p>If WIDTH is greater than 0, the string is completed with trailing blank characters in order to match the absolute value of WIDTH.</p> <p>If DIGITS is 0 then neither decimal part nor point are added.</p> <p>If DIGITS is greater than 0, the corresponding number of decimal digits are added. '0' digits are added if necessary.</p> <p>If the value is too long for the specified width, then the string is filled with '*' characters.</p>	
Examples	<pre>Q := NUM_TO_STRING (123.4, 8, 2); (* Q is ' 123.40' *) Q := NUM_TO_STRING (123.4, -8, 2); (* Q is '123.40 ' *) Q := NUM_TO_STRING (1.333333, 0, 2); (* Q is '1.33' *) Q := NUM_TO_STRING (1234, 3, 0); (* Q is '***' *)</pre>	



2.13.11. BCD_TO_BIN

Function Converts a BCD (Binary Coded Decimal) value to a binary value

Inputs IN : DINT Integer value in BCD

Outputs Q : DINT Value converted to integer

or 0 if IN is not a valid positive BCD value

Truth table

IN	Q
-2	0 (invalid)
0	0
16 (16#10)	10
15 (16#0F)	0 (invalid)

The data in this truth table is examples.

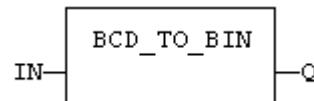
Remarks

The input must be positive and must represent a valid BCD value. In LD language, the operation is executed only if the input rung (EN) is *TRUE*. The output rung (ENO) keeps the same value as the input rung. In IL, the input must be loaded in the current result before calling the function.

ST Language

Q := BCD_TO_BIN (IN);

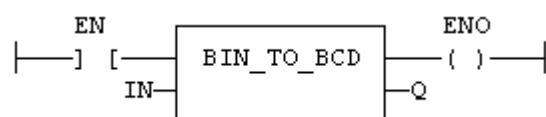
FBD Language



LD Language

The conversion is executed only if EN is *TRUE*.

ENO keeps the same value as EN.



IL Language

Op1: LD IN

BCD_TO_BIN

ST Q

2.13.12. BIN_TO_BCD

Function Converts a binary value to a BCD (Binary Coded Decimal) value

Inputs IN : DINT Integer value

Outputs Q : DINT Value converted to BCD
or 0 if IN is less than 0

Truth table

IN	Q
-2	0 (invalid)
0	0
10	16 (16#10)
22	34 (16#34)

The data in this truth table is examples.

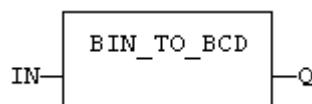
Remarks

The input must be positive. In LD language, the operation is executed only if the input rung (EN) is *TRUE*. The output rung (ENO) keeps the same value as the input rung. In IL, the input must be loaded in the current result before calling the function.

ST Language

`Q := BIN_TO_BCD (IN);`

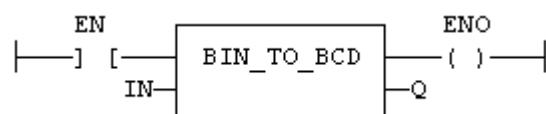
FBD Language



LD Language

The conversion is executed only if EN is *TRUE*.

ENO keeps the same value as EN.



IL Language

`Op1: LD IN
BIN_TO_BCD
ST Q`



2.14. Selectors

Below are the standard functions that perform data selection:

Function	Description
SEL	2 integer inputs
MUX4	4 integer input
MUX8	8 integer input



2.14.1. MUX4

Function Select one of the inputs - 4 inputs

Inputs SELECT : DINT Selection command

IN1 : ANY First input

IN2 : ANY Second input

:

IN4 : ANY Last input

Outputs Q : ANY IN1 or IN2 ... or IN4 depending on *SELECT* (see truth table)

Truth table

SELECT	Q
0	IN1
1	IN2
2	IN3
3	IN4
other	0

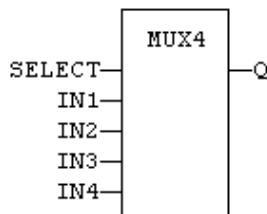
The data in this truth table is examples.

Remarks

In LD language, the input rung (EN) enables the selection. The output rung keeps the same state as the input rung. In IL language, the first parameter (selector) must be loaded in the current result before calling the function. Other inputs are operands of the function, separated by commas.

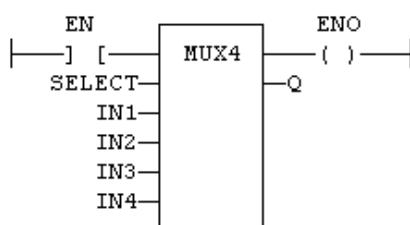
ST Language `Q := MUX4 (SELECT, IN1, IN2, IN3, IN4);`

FBD Language



LD Language The selection is performed only if EN is *TRUE*.

ENO has the same value as EN.



IL Language

`Op1: LD SELECT
MUX4 IN1, IN2, IN3, IN4`



ST Q



2.14.2. MUX8

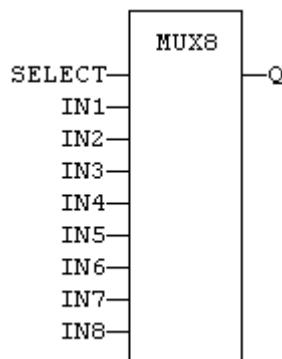
Function	Select one of the inputs - 8 inputs	
Inputs	SELECT : DINT Selection command. IN1 : ANY First input IN2 : ANY Second input : IN8 : ANY Last input	
Outputs	Q : ANY	IN1 or IN2 or IN8 depending on SELECT (see truth table)

Truth table	SELECT	Q
0		IN1
1		IN2
2		IN3
3		IN4
4		IN5
5		IN6
6		IN7
7		IN8
other		0

Remarks In LD language, the input rung (EN) enables the selection. The output rung keeps the same state as the input rung. In IL language, the first parameter (selector) must be loaded in the current result before calling the function. Other inputs are operands of the function, separated by commas.

ST Language `Q := MUX8 (SELECT, IN1, IN2, IN3, IN4, IN5, IN6, IN7, IN8);`

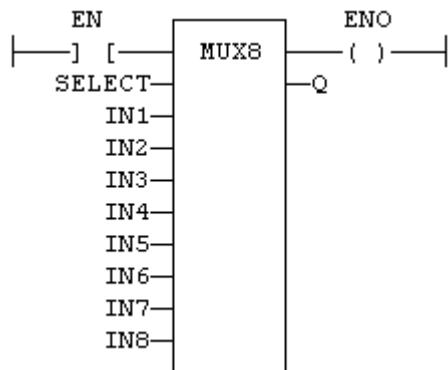
FBD Language





LD Language The selection is performed only if EN is *TRUE*.

ENO has the same value as EN.



IL Language Op1: LD SELECT

MUX8 IN1, IN2, IN3, IN4, IN5, IN6, IN7, IN8

ST Q



2.14.3. SEL

Function Select one of the inputs - 2 inputs

Inputs SELECT : BOOL Selection command

IN1 : ANY First input

IN2 : ANY Second input

Outputs Q : ANY IN1 if SELECT is FALSE; IN2 if SELECT is TRUE

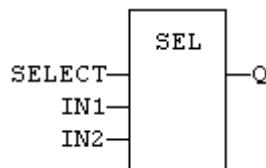
Truth table

SELECT	Q
0	IN1
1	IN2

Remarks In LD language, the selector command is the input rung. The output rung keeps the same state as the input rung. In IL language, the first parameter (selector) must be loaded in the current result before calling the function. Other inputs are operands of the function, separated by commas.

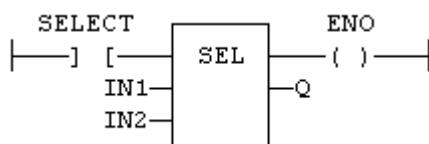
ST Language Q := SEL (SELECT, IN1, IN2);

FBD Language



LD Language The input rung is the selector.

ENO has the same value as SELECT.



IL Language Op1: LD SELECT

SEL IN1, IN2

ST Q



2.15. Registers

Below are the standard functions for managing 8 bit to 32 bit registers:

Function	Description
SHL	shift left
SHR	shift right
ROL	rotation left
ROR	rotation right

Below are advanced functions for register manipulation:

Function	Description
MBShift	multibyte shift / rotate

The following functions enable bit to bit operations on a 8 bit to 32 bit integers:

Function	Description
AND_MASK	Boolean AND
OR_MASK	Boolean OR
XOR_MASK	exclusive OR
NOT_MASK	Boolean negation

The following functions enable to pack/unpack 8, 16 and 32 bit registers

Function	Description
LOBYTE	Get the lowest byte of a word
HIBYTE	Get the highest byte of a word
LOWORD	Get the lowest word of a double word
HIWORD	Get the highest word of a double word
MAKEWORD	Pack bytes to a word
MAKEDWORD	Pack words to a double word
PACK8	Pack bits in a byte
UNPACK8	Extract bits from a byte



The following functions provide bit access in 8 bit to 32 bit integers:

Function	Description
SETBIT	Set a bit in a register
TESTBIT	Test a bit of a register

The following functions are maintained for compatibility, but you should use the functions listed above:

AND_DINT, AND_UDINT, AND_DWORD, NOT_DINT, NOT_UDINT, NOT_DWORD
OR_DINT, OR_UDINT, OR_DWORD, XOR_DINT, XOR_UDINT, XOR_DWORD
AND_INT, AND_UINT, AND_WORD, NOT_INT, NOT_UINT, NOT_WORD
OR_INT, OR_UINT, OR_WORD, XOR_INT, XOR_UINT, XOR_WORD
AND_SINT, AND_USINT, AND_BYTE, NOT_SINT, NOT_USINT, NOT_BYTE
OR_SINT, OR_USINT, OR_BYTE, XOR_SINT, XOR_USINT, XOR_BYTE
rolw, rorw, shlw, shrw, rolb, rorb, shlb, shrb
ROL_DINT, ROR_DINT, SHL_DINT, SHR_DINT
ROL_UDINT, ROR_UDINT, SHL_UDINT, SHR_UDINT
ROL_DWORD, ROR_DWORD, SHL_DWORD, SHR_DWORD
ROL_INT, ROR_INT, SHL_INT, SHR_INT
ROL_UINT, ROR_UINT, SHL_UINT, SHR_UINT
ROL_WORD, ROR_WORD, SHL_WORD, SHR_WORD
ROL_SINT, ROR_SINT, SHL_SINT, SHR_SINT
ROL_USINT, ROR_USINT, SHL_USINT, SHR_USINT
ROL_BYTE, ROR_BYTE, SHL_BYTE, SHR_BYTE



2.15.1. AND_MASK

Function Performs a bit to bit AND between two integer values

Inputs IN : ANY First input

MSK : ANY Second input (AND mask)

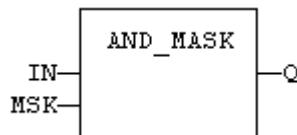
Outputs Q : ANY AND mask between IN and MSK inputs

Remarks Arguments can be signed or unsigned integers from 8 to 32 bits.

In LD language, the input rung (EN) enables the operation, and the output rung keeps the same value as the input rung. In IL language, the first parameter (IN) must be loaded in the current result before calling the function. The other input is the operands of the function.

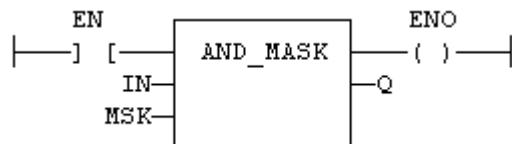
ST Language `Q := AND_MASK (IN, MSK);`

FBD Language



LD Language The function is executed only if EN is *TRUE*.

ENO is equal to EN:



IL Language `Op1: LD IN`

`AND_MASK MSK`

`ST Q`



2.15.2. HIBYTE

Function Get the most significant byte of a word

Inputs IN : UINT 16 bit register

Outputs Q : USINT Most significant byte

Remarks In LD language, the operation is executed only if the input rung (EN) is *TRUE*. The output rung (ENO) keeps the same value as the input rung. In IL, the input must be loaded in the current result before calling the function.

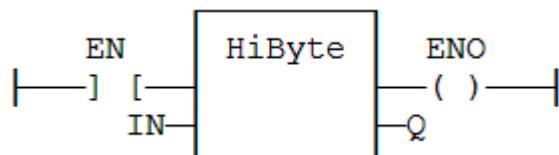
ST Language `Q := HIBYTE (IN);`

FBD Language



LD Language The function is executed only if EN is *TRUE*.

ENO keeps the same value as EN:



IL Language
`Op1: LD IN
HIBYTE
ST Q`

2.15.3. LOBYTE

Function Get the least significant byte of a word

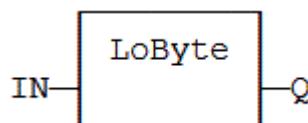
Inputs IN : UINT 16 bit register

Outputs Q : USINT Lowest significant byte

Remarks In LD language, the operation is executed only if the input rung (EN) is *TRUE*. The output rung (ENO) keeps the same value as the input rung. In IL, the input must be loaded in the current result before calling the function.

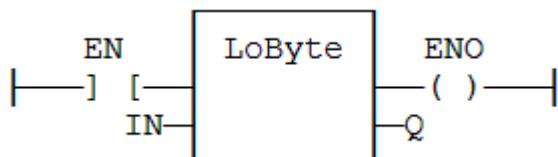
ST Language `Q := LOBYTE (IN);`

FBD Language



LD Language The function is executed only if EN is *TRUE*.

ENO keeps the same value as EN:



IL Language
`Op1: LD IN
 LOBYTE
 ST Q`

2.15.4. HIWORD

Function Get the most significant word of a double word

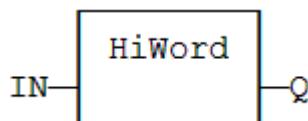
Inputs IN : UDINT 32 bit register

Outputs Q : UINT Most significant word

Remarks In LD language, the operation is executed only if the input rung (EN) is *TRUE*. The output rung (ENO) keeps the same value as the input rung. In IL, the input must be loaded in the current result before calling the function.

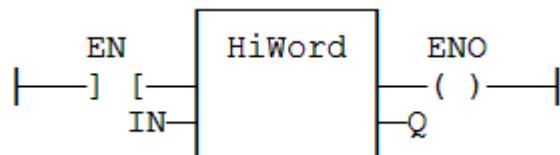
ST Language `Q := HIWORD (IN);`

FBD Language



LD Language The function is executed only if EN is *TRUE*.

ENO keeps the same value as EN:



IL Language

Op1: LD IN

HIWORD

ST Q

2.15.5. LOWORD

Function Get the least significant word of a double word

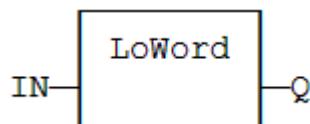
Inputs IN : UDINT 32 bit register

Outputs Q : UINT Least significant word

Remarks In LD language, the operation is executed only if the input rung (EN) is *TRUE*. The output rung (ENO) keeps the same value as the input rung. In IL, the input must be loaded in the current result before calling the function.

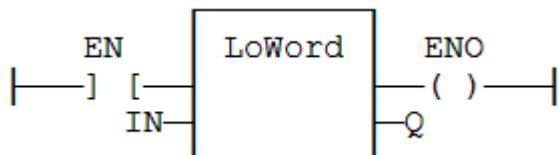
ST Language `Q := LOWORD (IN);`

FBD Language



LD Language The function is executed only if EN is *TRUE*.

ENO keeps the same value as EN:



IL Language

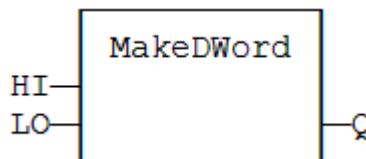
`Op1: LD IN
LOWORD
ST Q`



2.15.6. MAKEDWORD

Function	Builds a double word as the concatenation of two words	
Inputs	HI : USINT	Highest significant word
	LO : USINT	Lowest significant word
Outputs	Q : UINT	32 bit register
Remarks	In LD language, the operation is executed only if the input rung (EN) is <i>TRUE</i> . The output rung (ENO) keeps the same value as the input rung. In IL, the first input must be loaded in the current result before calling the function.	
ST Language	<code>Q := MAKEDWORD (HI, LO);</code>	

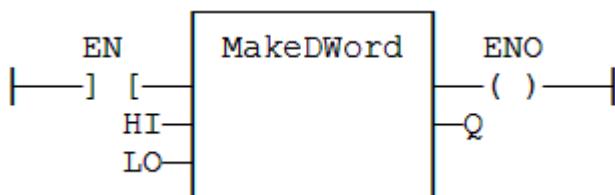
FBD Language



LD Language

The function is executed only if EN is *TRUE*.

ENO keeps the same value as EN:



IL Language

```
Op1: LD    HI  
      MAKEDWORD LO  
      ST     Q
```



2.15.7. MAKEWORD

Function Builds a word as the concatenation of two bytes

Inputs HI : USINT Highest significant byte

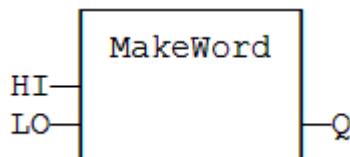
LO : USINT Lowest significant byte

Outputs Q : UINT 16 bit register

Remarks In LD language, the operation is executed only if the input rung (EN) is *TRUE*. The output rung (ENO) keeps the same value as the input rung. In IL, the first input must be loaded in the current result before calling the function.

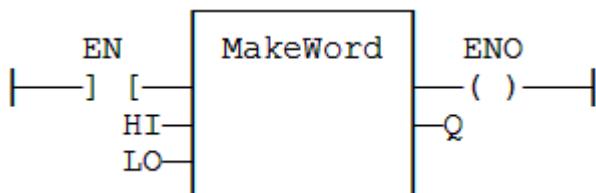
ST Language `Q := MAKEWORD (HI, LO);`

FBD Language



LD Language The function is executed only if EN is *TRUE*.

ENO keeps the same value as EN:



IL Language

```
Op1: LD    HI
      MAKEWORD LO
      ST    Q
```

2.15.8. MBSHIFT

Function Multibyte shift / rotate

Inputs	Buffer : SINT/USINT	Array of bytes
	Pos : DINT	Base position in the array
	NbByte : DINT	Number of bytes to be shifted/rotated
	NbShift : DINT	Number of shifts or rotations
	ToRight : BOOL	<i>TRUE</i> for right / <i>FALSE</i> for left
	Rotate : BOOL	<i>TRUE</i> for rotate / <i>FALSE</i> for shift
	InBit : BOOL	Bit to be introduced in a shift

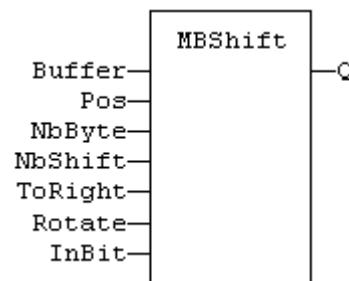
Outputs Q : BOOL *TRUE* if successful

Remarks Use the *ToRight* argument to specify a shift to the left (*FALSE*) or to the right (*TRUE*). Use the *Rotate* argument to specify either a shift (*FALSE*) or a rotation (*TRUE*). In case of a shift, the *InBit* argument specifies the value of the bit that replaces the last shifted bit.

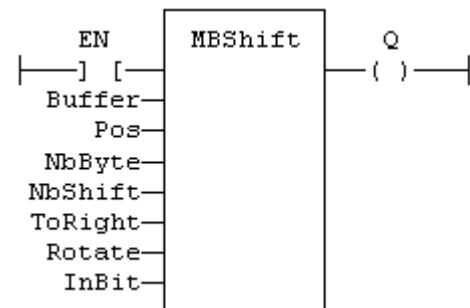
In LD language, the rung input (EN) validates the operation. The rung output is the result (Q).

ST Language `Q := MBShift (Buffer, Pos, NbByte, NbShift, ToRight, Rotate, InBit);`

FBD Language



LD Language The function is called only if EN is *TRUE*.



IL Language Not available.



2.15.9. NOT_MASK

Function Performs a bit to bit negation of an integer value

Inputs IN : ANY Integer input

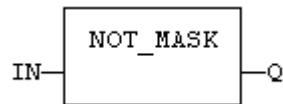
Outputs Q : ANY Bit to bit negation of the input

Remarks Arguments can be signed or unsigned integers from 8 to 32 bits.

In LD language, the input rung (EN) enables the operation, and the output rung keeps the same value as the input rung. In IL language, the parameter (IN) must be loaded in the current result before calling the function.

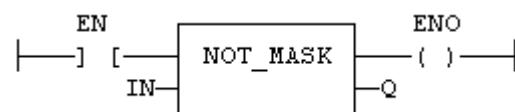
ST Language `Q := NOT_MASK (IN);`

FBD Language



LD Language The function is executed only if EN is *TRUE*.

ENO is equal to EN:



IL Language

`Op1: LD IN
NOT_MASK
ST Q`



2.15.10. OR_MASK

Function Performs a bit to bit OR between two integer values

Inputs IN : ANY First input

 MSK : ANY Second input (OR mask)

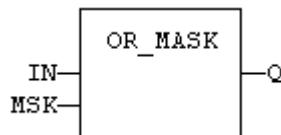
Outputs Q : ANY OR mask between IN and MSK inputs

Remarks Arguments can be signed or unsigned integers from 8 to 32 bits.

In LD language, the input rung (EN) enables the operation, and the output rung keeps the same value as the input rung. In IL language, the first parameter (IN) must be loaded in the current result before calling the function. The other input is the operands of the function.

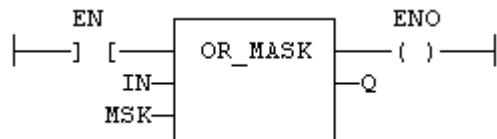
ST Language `Q := OR_MASK (IN, MSK);`

FBD Language



LD Language The function is executed only if EN is *TRUE*.

ENO is equal to EN:



IL Language `Op1: LD IN`

`OR_MASK MSK`

`ST Q`



2.15.11. PACK8

Function Builds a byte with bits

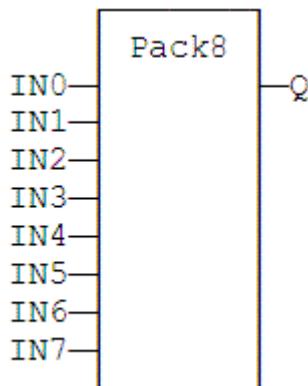
Inputs IN0 : BOOL Less significant bit
IN7 : BOOL Most significant bit

Outputs Q : USINT Byte built with input bits

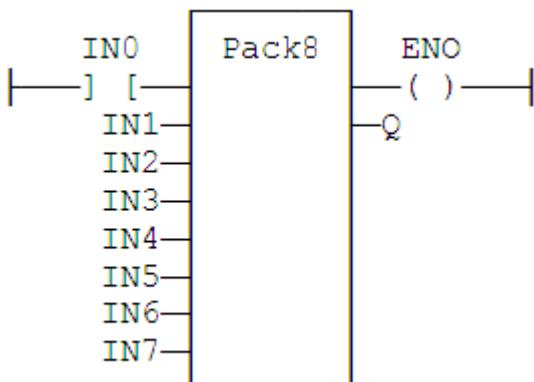
Remarks In LD language, the input rung is the IN0 input. The output rung (ENO) keeps the same value as the input rung. In IL, the input must be loaded in the current result before calling the function.

ST Language `Q := PACK8 (IN0, IN1, IN2, IN3, IN4, IN5, IN6, IN7);`

FBD Language



LD Language ENO keeps the same value as EN:



IL Language Op1: LD IN0

`PACK8 IN1, IN2, IN3, IN4, IN5, IN6, IN7`

`ST Q`



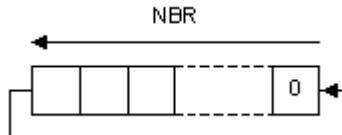
2.15.12. ROL

Function Rotate bits of a register to the left

Inputs IN : ANY register
 NBR : DINT Number of rotations (each rotation is 1 bit)

Outputs Q : ANY Rotated register

Diagram

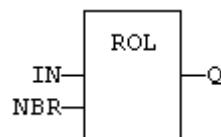


Remarks Arguments can be signed or unsigned integers from 8 to 32 bits.

In LD language, the input rung (EN) enables the operation, and the output rung keeps the state of the input rung. In IL language, the first input must be loaded before the function call. The second input is the operand of the function.

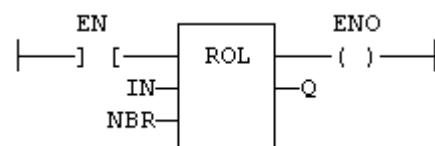
ST Language `Q := ROL (IN, NBR);`

FBD Language



LD Language The rotation is executed only if EN is *TRUE*.

ENO has the same value as EN:



IL Language `Op1: LD IN
 ROL NBR
 ST Q`

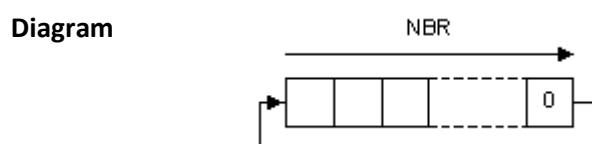
2.15.13. ROR

Function Rotate bits of a register to the right

Inputs IN : ANY register

 NBR : ANY Number of rotations (each rotation is 1 bit)

Outputs Q : ANY Rotated register

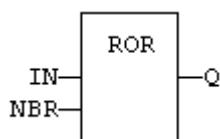


Remarks Arguments can be signed or unsigned integers from 8 to 32 bits.

In LD language, the input rung (EN) enables the operation, and the output rung keeps the state of the input rung. In IL language, the first input must be loaded before the function call. The second input is the operand of the function.

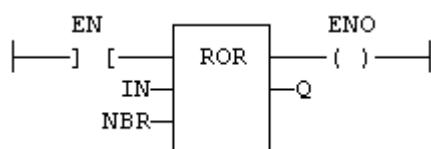
ST Language `Q := ROR (IN, NBR);`

FBD Language



LD Language The rotation is executed only if EN is *TRUE*.

ENO has the same value as EN:



IL Language `Op1: LD IN`

`ROR NBR`

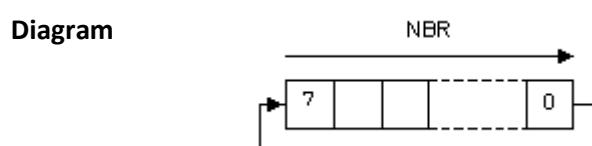
`ST Q`

2.15.14. RORb / ROR_SINT / ROR_USINT / ROR_BYTE

Function Rotate bits of a register to the right

Inputs IN : SINT 8 bit register
 NBR : SINT Number of rotations (each rotation is 1 bit)

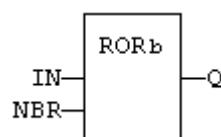
Outputs Q : ANY Rotated register



Remarks In LD language, the input rung (EN) enables the operation, and the output rung keeps the state of the input rung. In IL language, the first input must be loaded before the function call. The second input is the operand of the function.

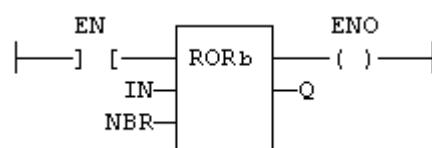
ST Language `Q := RORb (IN, NBR);`

FBD Language



LD Language The rotation is executed only if EN is *TRUE*.

ENO has the same value as EN:



IL Language `Op1: LD IN
 RORb NBR
 ST Q`

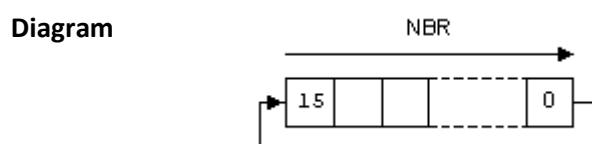


2.15.15. RORw / ROR_INT / ROR_UINT / ROR_WORD

Function Rotate bits of a register to the right

Inputs IN : INT 16 bit register
 NBR : INT Number of rotations (each rotation is 1 bit)

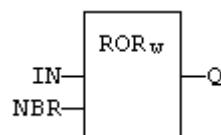
Outputs Q : INT Rotated register



Remarks In LD language, the input rung (EN) enables the operation, and the output rung keeps the state of the input rung. In IL language, the first input must be loaded before the function call. The second input is the operand of the function.

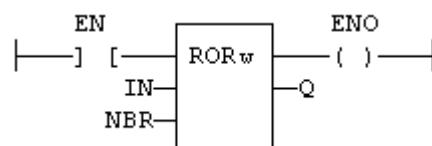
ST Language `Q := RORw (IN, NBR);`

FBD Language



LD Language The rotation is executed only if EN is *TRUE*.

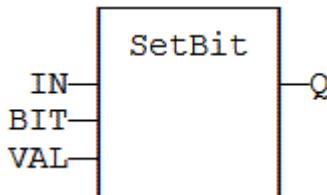
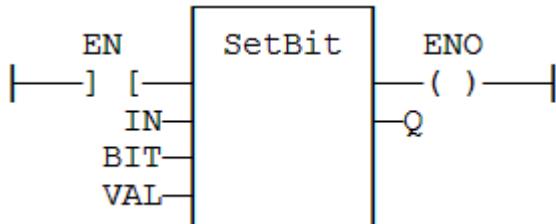
ENO has the same value as EN:



IL Language `Op1: LD IN
 RORw NBR
 ST Q`



2.15.16. SETBIT

Function	Set a bit in an integer register
Inputs	IN : ANY 8 to 32 bit integer register BIT : DINT Bit number (0 = less significant bit) VAL : BOOL Bit value to apply
Outputs	Q : ANY Modified register
Remarks	Types LINT, REAL, LREAL, TIME and STRING are not supported for IN and Q. IN and Q must have the same type. In case of invalid arguments (bad bit number or invalid input type) the function returns the value of IN without modification. In LD language, the operation is executed only if the input rung (EN) is TRUE. The output rung (ENO) keeps the same value as the input rung.
ST Language	<code>Q := SETBIT (IN, BIT, VAL);</code>
FBD Language	
LD Language	The function is executed only if EN is <i>TRUE</i> . ENO keeps the same value as EN: 
IL Language	<code>Not available.</code>

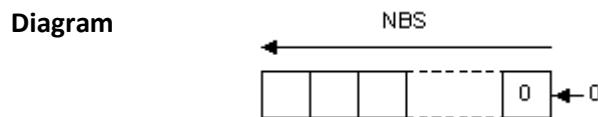


2.15.17. SHL

Function Shift bits of a register to the left

Inputs IN : ANY register
NBS : ANY Number of shifts (each shift is 1 bit)

Outputs Q : ANY Shifted register

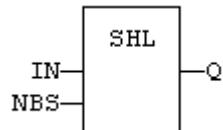


Remarks Arguments can be signed or unsigned integers from 8 to 32 bits.

In LD language, the input rung (EN) enables the operation, and the output rung keeps the state of the input rung. In IL language, the first input must be loaded before the function call. The second input is the operand of the function.

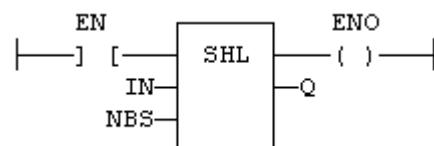
ST Language `Q := SHL (IN, NBS);`

FBD Language



LD Language The shift is executed only if EN is TRUE.

ENO has the same value as EN:



IL Language
`Op1: LD IN
SHL NBS
ST Q`



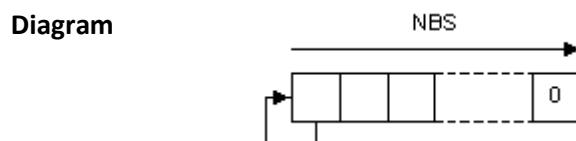
2.15.18. SHR

Function Shift bits of a register to the right

Inputs IN : ANY register

NBS : ANY Number of shifts (each shift is 1 bit)

Outputs Q : ANY Shifted register

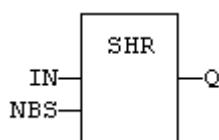


Remarks Arguments can be signed or unsigned integers from 8 to 32 bits.

In LD language, the input rung (EN) enables the operation, and the output rung keeps the state of the input rung. In IL language, the first input must be loaded before the function call. The second input is the operand of the function.

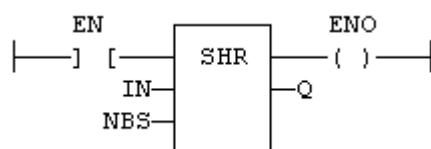
ST Language `Q := SHR (IN, NBS);`

FBD Language



LD Language The shift is executed only if EN is TRUE.

ENO has the same value as EN:



L Language `Op1: LD IN
SHR NBS
ST Q`

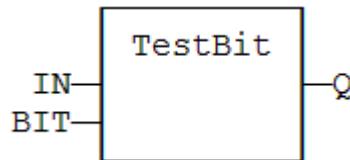


2.15.19. TESTBIT

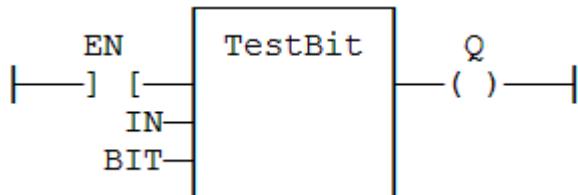
Function	Test a bit of an integer register	
Inputs	IN : ANY	8 to 32 bit integer register
	BIT : DINT	Bit number (0 = less significant bit)
Outputs	Q : BOOL	Bit value
Remarks	<p>Types LINT, REAL, LREAL, TIME and STRING are not supported for IN and Q. IN and Q must have the same type. In case of invalid arguments (bad bit number or invalid input type) the function returns <i>FALSE</i>.</p> <p>In LD language, the operation is executed only if the input rung (EN) is <i>TRUE</i>. The output rung is the output of the function.</p>	

ST Language `Q := TESTBIT (IN, BIT);`

FBD Language



LD Language The function is executed only if EN is *TRUE*.



IL Language Not available.

2.15.20. UNPACK8

Function Block Extract bits of a byte

Inputs IN : USINT 8 bit register

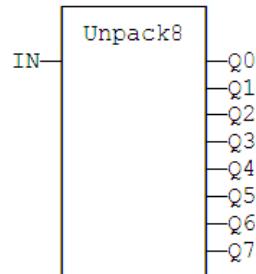
Outputs Q0 : BOOL Less significant bit
Q7 : BOOL Most significant bit

Remarks In LD language, the output rung is the Q0 output. The operation is executed only in the input rung (EN) is TRUE.

ST Language MyUnpack is a declared instance of the UNPACK8 function block.

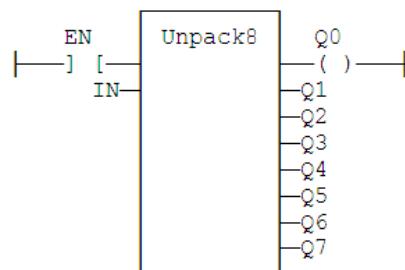
```
MyUnpack (IN );
Q0 := MyUnpack.Q0;
Q1 := MyUnpack.Q1;
Q2 := MyUnpack.Q2;
Q3 := MyUnpack.Q3;
Q4 := MyUnpack.Q4;
Q5 := MyUnpack.Q5;
Q6 := MyUnpack.Q6;
Q7 := MyUnpack.Q7;
```

FBD Language



LD Language

The function is executed only if EN is TRUE.



IL Language

MyUnpack is a declared instance of the UNPACK8 function block.

```
Op1: CAL MyUnpack (IN )
LD MyUnpack.Q0
ST Q0
(* ... *)
LD MyUnpack.Q7
ST Q7
```

2.15.21. XOR_MASK

Function	Performs a bit to bit exclusive OR between two integer values
Inputs	IN : ANY First input MSK : ANY Second input (XOR mask)
Outputs	Q : ANY Exclusive OR mask between IN and MSK inputs
Remarks	Arguments can be signed or unsigned integers from 8 to 32 bits. In LD language, the input rung (EN) enables the operation, and the output rung keeps the same value as the input rung. In IL language, the first parameter (IN) must be loaded in the current result before calling the function. The other input is the operands of the function.
ST Language	<code>Q := XOR_MASK (IN, MSK);</code>
FBD Language	
LD Language	The function is executed only if EN is <i>TRUE</i> . ENO is equal to EN
IL Language	<code>Op1: LD IN XOR_MASK MSK ST Q</code>



2.16. Counters

Below are the standard blocks for managing counters:

Block	Description
CTU	Up counter
CTD	Down counter
CTUD	Up / Down counter

2.16.1. CTD / CTDr

Function Block Down counter

Inputs CD : BOOL Enable counting. Counter is decreased on each call when CU is *TRUE*.

LOAD : BOOL Re-load command.

Counter is set to PV when called with LOAD to *TRUE*.

PV : DINT Programmed maximum value.

Outputs Q : BOOL *TRUE* when counter is empty, i.e. when CV = 0.

CV : DINT Current value of the counter.

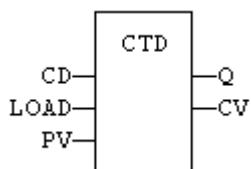
Remarks The counter is empty (CV = 0) when the application starts. The counter does not include a pulse detection for CD input. Use R_TRIG or F_TRIG function block for counting pulses of CD input signal. In LD language, CD is the input rung. The output rung is the Q output.

CTUr, CTDr, CTUDr function blocks operate exactly as other counters, except that all Boolean inputs (CU, CD, RESET, LOAD) have an implicit rising edge detection included. Note that these counters may be not supported on some target systems.

ST Language MyCounter is a declared instance of CTD function block.

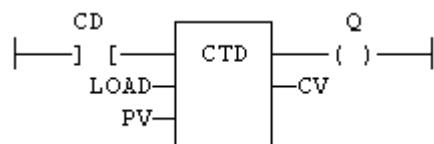
```
MyCounter (CD, LOAD, PV);  
Q := MyCounter.Q;  
CV := MyCounter.CV;
```

FBD Language



LD Language The function is executed only if EN is *TRUE*.

ENO is equal to EN



IL Language MyCounter is a declared instance of CTD function block.

```
Op1: CAL MyCounter (CD, LOAD, PV)  
LD MyCounter.Q  
ST Q  
LD MyCounter.CV  
ST CV
```



2.16.2. CTU / CTUr

Function Block Up counter

Inputs CU : BOOL Enable counting. Counter is increased on each call when CU is *TRUE*.

RESET : BOOL Reset command.

Counter is reset to 0 when called with RESET to *TRUE*.

PV : DINT Programmed maximum value.

Outputs Q : BOOL *TRUE* when counter is full, i.e. when CV = PV.

CV : DINT Current value of the counter.

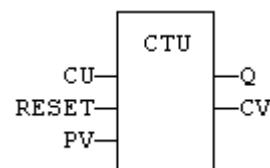
Remarks The counter is empty (CV = 0) when the application starts. The counter does not include a pulse detection for CU input. Use R_TRIG or F_TRIG function block for counting pulses of CU input signal. In LD language, CU is the input rung. The output rung is the Q output.

CTUr, CTDr, CTUDr function blocks operate exactly as other counters, except that all Boolean inputs (CU, CD, RESET, LOAD) have an implicit rising edge detection included. Note that these counters may be not supported on some target systems.

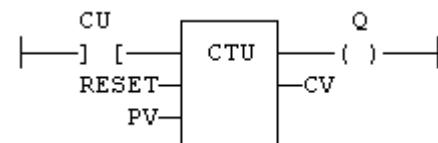
ST Language MyCounter is a declared instance of CTU function block.

```
MyCounter(CU, RESET, PV);  
Q := MyCounter.Q;  
CV := MyCounter.CV;
```

FBD Language



LD Language



IL Language

MyCounter is a declared instance of CTU function block.

```
Op1: CAL MyCounter(CU, RESET, PV)  
LD MyCounter.Q  
ST Q  
LD MyCounter.CV  
ST CV
```



2.16.3. CTUD / CTUDr

Function Block Up/down counter

Inputs	CU : BOOL	Enable counting. Counter is increased on each call when CU is <i>TRUE</i> .
	CD : BOOL	Enable counting. Counter is decreased on each call when CD is <i>TRUE</i> .
	RESET : BOOL	Reset command. Counter is reset to 0 called with RESET to <i>TRUE</i> .
	LOAD : BOOL	Re-load command. Counter is set to PV when called with LOAD to <i>TRUE</i> .
	PV : DINT	Programmed maximum value.
Outputs	QU : BOOL	<i>TRUE</i> when counter is full, i.e. when CV = PV.
	QD : BOOL	<i>TRUE</i> when counter is empty, i.e. when CV = 0.
	CV : DINT	Current value of the counter.

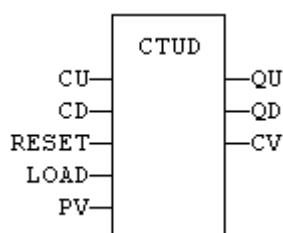
Remarks The counter is empty (CV = 0) when the application starts. The counter does not include a pulse detection for CU and CD inputs. Use R_TRIG or F_TRIG function blocks for counting pulses of CU or CD input signals. In LD language, CU is the input rung. The output rung is the QU output.

CTUr, CTDr, CTUDr function blocks operate exactly as other counters, except that all Boolean inputs (CU, CD, RESET, LOAD) have an implicit rising edge detection included. Note that these counters may be not supported on some target systems.

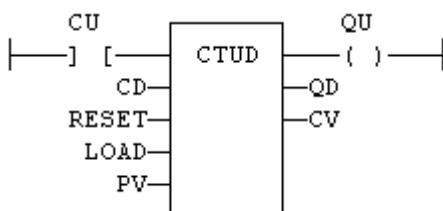
ST Language MyCounter is a declared instance of CTUD function block.

```
MyCounter (CU, CD, RESET, LOAD, PV);  
QU := MyCounter.QU;  
QD := MyCounter.QD;  
CV := MyCounter.CV;
```

FBD Language



LD Language





IL Language

MyCounter is a declared instance of CTUD function block.

Op1: CAL MyCounter (CU, CD, RESET, LOAD, PV)

```
LD  MyCounter.QU
ST  QU
LD  MyCounter.QD
ST  QD
LD  MyCounter.CV
ST  CV
```



2.17. Timers

Below are the standard functions for managing timers:

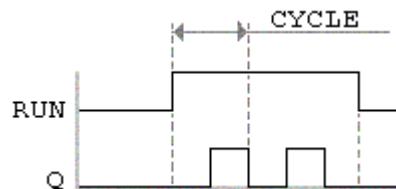
Function	Effect
TON	On timer
TOF	Off timer
TP	Pulse timer
BLINK	Blinker
BLINKA	Asymmetric blinker
PLS	Pulse signal generator
TMU	Up-counting stop watch
TMD	Down-counting stop watch

2.17.1. BLINK

Function	Flashes	
Inputs	RUN : BOOL	Enabling command
	CYCLE : TIME	Blinking period

Outputs	Q : BOOL	Output blinking signal
----------------	----------	------------------------

Time Diagram

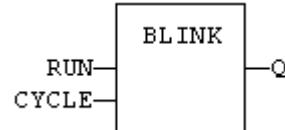


Remarks The output signal is *FALSE* when the RUN input is *FALSE*. The CYCLE input is the complete period of the blinking signal. In LD language, the input rung is the IN command. The output rung is the Q output signal.

ST Language MyBlinker is a declared instance of BLINK function block.

```
MyBlinker (RUN, CYCLE );  
Q := MyBlinker.Q;
```

FBD Language



LD Language



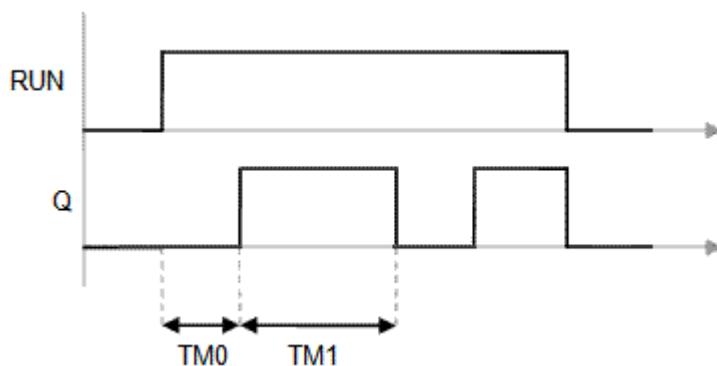
IL Language

MyBlinker is a declared instance of BLINK function block.

```
Op1: CAL MyBlinker (RUN, CYCLE )  
LD MyBlinker.Q  
ST Q
```

2.17.2. BLINKA

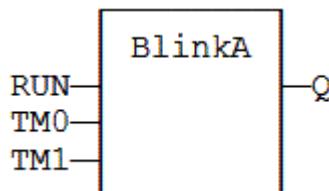
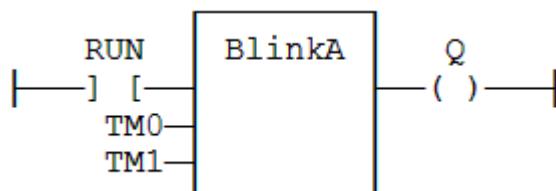
Function	Asymmetric flashing	
Inputs	RUN : BOOL	Enabling command
	TM0 : TIME	Duration of <i>FALSE</i> state on output
	TM1 : TIME	Duration of <i>TRUE</i> state on output
Outputs	Q : BOOL	Output blinking signal

Time Diagram

Remarks The output signal is *FALSE* when the RUN input is *FALSE*. In LD language, the input rung is the IN command. The output rung is the Q output signal.

ST Language MyBlinker is a declared instance of BLINKA function block.

```
MyBlinker (RUN, TM0, TM1 );  
Q := MyBlinker.Q;
```

FBD Language**LD Language****IL Language**

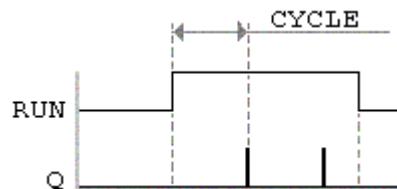
MyBlinker is a declared instance of BLINKA function block.

```
Op1: CAL MyBlinker (RUN, TM0, TM1 )  
LD MyBlinker.Q  
ST Q
```

2.17.3. PLS

Function	Pulse signal generator	
Inputs	RUN : BOOL	Enabling command
	CYCLE : TIME	Signal period

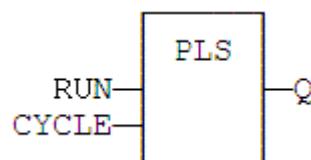
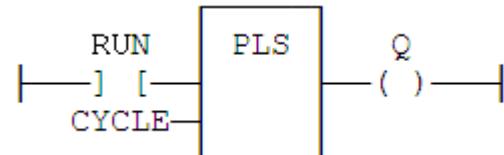
Outputs	Q : BOOL	Output pulse signal
----------------	----------	---------------------

Time Diagram

Remarks On every period, the output is set to TRUE during one cycle only. In LD language, the input rung is the IN command. The output rung is the Q output signal.

ST Language MyPLS is a declared instance of PLS function block:

```
MyPLS (RUN, CYCLE );  
Q := MyPLS.Q;
```

FBD Language**LD Language****IL Language**

MyPLS is a declared instance of PLS function block:

```
Op1: CAL MyPLS (RUN, CYCLE )  
LD MyPLS.Q  
ST Q
```

2.17.4. TMD

Function Counting the stop watch downwards

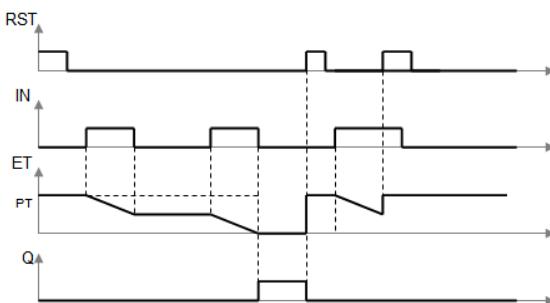
Inputs

IN : BOOL	The time counts when this input is <i>TRUE</i>
RST : BOOL	Timer is reset to PT when this input is <i>TRUE</i>
PT : TIME	Programmed time

Outputs

Q : BOOL	Timer elapsed output signal
ET : TIME	Elapsed time

Time Diagram

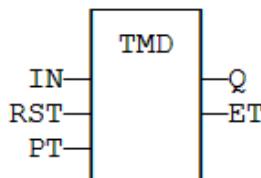


Remarks The timer counts up when the IN input is *TRUE*. It stops when the programmed time is elapsed. The timer is reset when the RST input is *TRUE*. It is not reset when IN is false.

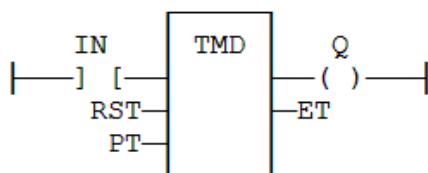
ST Language MyTimer is a declared instance of TMD function block:

```
MyTimer (IN, RST, PT);  
Q := MyTimer.Q;  
ET := MyTimer.ET;
```

FBD Language



LD Language

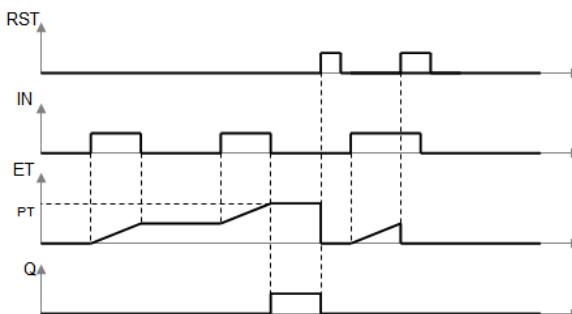
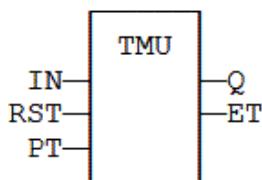
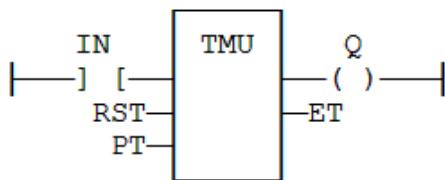


IL Language

MyTimer is a declared instance of TMD function block:

```
Op1: CAL MyTimer (IN, RST, PT)  
LD MyTimer.Q  
ST Q  
LD MyTimer.ET  
ST ET
```

2.17.5. TMU

Function	Counting the stop watch upwards
Inputs	IN : BOOL The time counts when this input is <i>TRUE</i> RST : BOOL Timer is reset to 0 when this input is <i>TRUE</i> PT : TIME Programmed time
Outputs	Q : BOOL Timer elapsed output signal ET : TIME Elapsed time
Time Diagram	
Remarks	The timer counts up when the IN input is <i>TRUE</i> . It stops when the programmed time is elapsed. The timer is reset when the RST input is <i>TRUE</i> . It is not reset when IN is false.
ST Language	MyTimer is a declared instance of TMU function block: <pre>MyTimer (IN, RST, PT); Q := MyTimer.Q; ET := MyTimer.ET;</pre>
FBD Language	
LD Language	
IL Language	MyTimer is a declared instance of TMU function block: <pre>Op1: CAL MyTimer (IN, RST, PT) LD MyTimer.Q ST Q LD MyTimer.ET ST ET</pre>

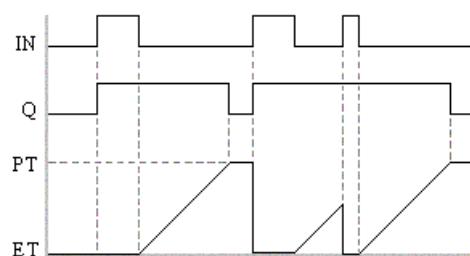
2.17.6. TOF / TOFR

Function Switch off timer

Inputs IN : BOOL Timer command
PT : TIME Programmed time
RST : BOOL Reset (TOFR only)

Outputs Q : BOOL Timer elapsed output signal
ET : TIME Elapsed time

Time Diagram



Remarks

The timer starts on a falling pulse of IN input. It stops when the elapsed time is equal to the programmed time. A rising pulse of IN input resets the timer to 0. The output signal is set to *TRUE* on when the IN input rises to *TRUE*, reset to *FALSE* when programmed time is elapsed.

TOFR is same as TOF but has an extra input for resetting the timer.

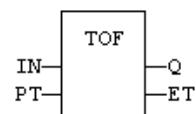
In LD language, the input rung is the IN command. The output rung is Q the output signal.

ST Language

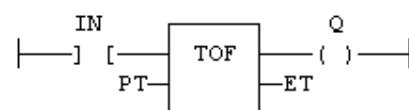
MyTimer is a declared instance of TOF function block.

```
MyTimer (IN, PT );  
Q := MyTimer.Q;  
ET := MyTimer.ET;
```

FBD Language



LD Language



IL Language

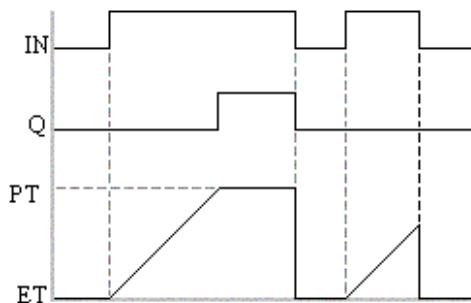
MyTimer is a declared instance of TOF function block:

```
Op1: CAL MyTimer (IN, PT )  
LD MyTimer.Q  
ST Q  
LD MyTimer.ET  
ST ET
```

2.17.7. TON

Function	Switch on timer	
Inputs	IN : BOOL	Timer command
	PT : TIME	Programmed time
Outputs	Q : BOOL	Timer elapsed output signal
	ET : TIME	Elapsed time

Time Diagram



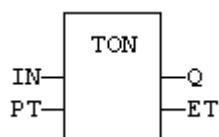
Remarks The timer starts on a rising pulse of IN input. It stops when the elapsed time is equal to the programmed time. A falling pulse of IN input resets the timer to 0. The output signal is set to *TRUE* when programmed time is elapsed, and reset to *FALSE* when the input command falls.

In LD language, the input rung is the IN command. The output rung is Q the output signal.

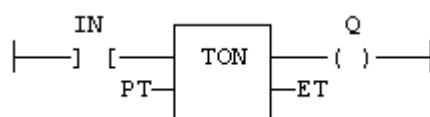
ST Language MyTimer is a declared instance of TON function block.

```
MyTimer (IN, PT);  
Q := MyTimer.Q;  
ET := MyTimer.ET;
```

FBD Language



LD Language



IL Language

MyTimer is a declared instance of TON function block:

```
Op1: CAL MyTimer (IN, PT)  
LD MyTimer.Q  
ST Q  
LD MyTimer.ET  
ST ET
```

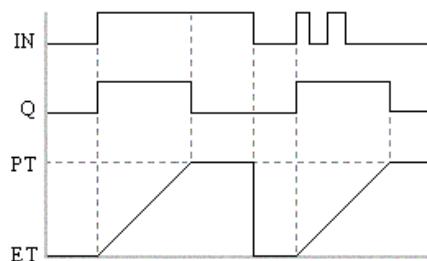
2.17.8. TP / TPR

Function Pulse timer

Inputs IN : BOOL Timer command
PT : TIME Programmed time
RST : BOOL Reset (TPR only)

Outputs Q : BOOL Timer elapsed output signal
ET : TIME Elapsed time

Time Diagram



Remarks

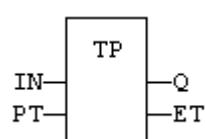
The timer starts on a rising pulse of IN input. It stops when the elapsed time is equal to the programmed time. A falling pulse of IN input resets the timer to 0, only if the programmed time is elapsed. All pulses of IN while the timer is running are ignored. The output signal is set to TRUE while the timer is running.
TPR is same as TP but has an extra input for resetting the timer.
In LD language, the input rung is the IN command. The output rung is Q the output signal.

ST Language

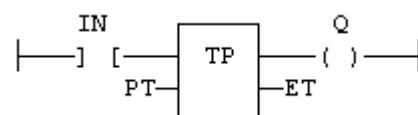
MyTimer is a declared instance of TP function block.

```
MyTimer (IN, PT);  
Q := MyTimer.Q;  
ET := MyTimer.ET;
```

FBD Language



LD Language



IL Language

MyTimer is a declared instance of TP function block.

```
Op1: CAL MyTimer (IN, PT)  
LD MyTimer.Q  
ST Q  
LD MyTimer.ET  
ST ET
```

2.18. Mathematic operations

Below are the standard functions that perform mathematic calculation:

Function	Effect
ABS	absolute value
TRUNC	integer part
LOG, LN	logarithm
POW, EXPT, EXP	power
SQRT	square root
SCALELIN	scaling - linear conversion

2.18.1. ABS

Function Returns the absolute value of the input

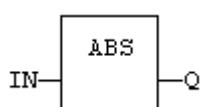
Inputs IN : ANY value

Outputs Q : ANY Result: absolute value of IN

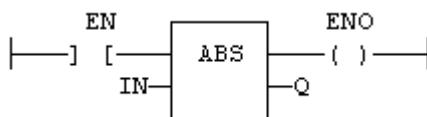
Remarks In LD language, the operation is executed only if the input rung (EN) is *TRUE*. The output rung (ENO) keeps the same value as the input rung. In IL, the input must be loaded in the current result before calling the function.

ST Language `Q := ABS (IN);`

FBD Language



LD Language



IL Language

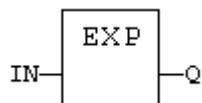
`Op1: LD IN
ABS
ST Q (* Q is: ABS (IN) *)`

2.18.2. EXP / EXPL

Function	Calculates the natural exponential of the input	
Inputs	IN : REAL/LREAL	Real value
Outputs	Q : REAL/LREAL	Result: natural exponential of IN
Remarks	In LD language, the operation is executed only if the input rung (EN) is <i>TRUE</i> . The output rung (ENO) keeps the same value as the input rung. In IL, the input must be loaded in the current result before calling the function.	

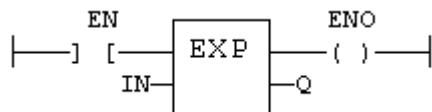
ST Language `Q := EXP (IN);`

FBD Language



LD Language The function is executed only if EN is *TRUE*.

ENO keeps the same value as EN.



IL Language Op1: LD IN

EXP

ST Q (* Q is: EXP (IN) *)



2.18.3. EXPT

Function Calculates a power

Inputs IN : REAL Real value

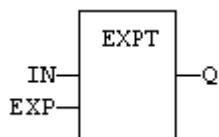
EXP : DINT Exponent

Outputs Q : REAL Result: IN at the 'EXP' power

Remarks In LD language, the operation is executed only if the input rung (EN) is *TRUE*. The output rung (ENO) keeps the same value as the input rung. In IL, the input must be loaded in the current result before calling the function. The exponent (second input of the function) must be the operand of the function.

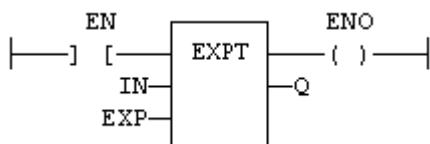
ST Language `Q := EXPT (IN, EXP);`

FBD Language



LD Language The function is executed only if EN is *TRUE*.

ENO keeps the same value as EN.



IL Language

`Op1: LD IN
EXPT EXP
ST Q (* Q is: (IN ** EXP) *)`

2.18.4. LOG

Function	Calculates the logarithm (base 10) of the input
Inputs	IN : REAL Real value
Outputs	Q : REAL Result: logarithm (base 10) of IN
Remarks	In LD language, the operation is executed only if the input rung (EN) is <i>TRUE</i> . The output rung (ENO) keeps the same value as the input rung. In IL, the input must be loaded in the current result before calling the function.
ST Language	Q := LOG (IN);
FBD Language	
LD Language	The function is executed only if EN is <i>TRUE</i> . ENO keeps the same value as EN.
IL Language	Op1: LD IN LOG ST Q (* Q is: LOG (IN) *)

2.18.5. LN

Function	Calculates the natural logarithm of the input
Inputs	IN : REAL/LREAL Real value
Outputs	Q : REAL/LREAL Result: natural logarithm of IN
Remarks	In LD language, the operation is executed only if the input rung (EN) is <i>TRUE</i> . The output rung (ENO) keeps the same value as the input rung. In IL, the input must be loaded in the current result before calling the function.
ST Language	<code>Q := LN (IN);</code>
FBD Language	
LD Language	The function is executed only if EN is <i>TRUE</i> . ENO keeps the same value as EN.
IL Language	<code>Op1: LD IN LN ST Q (* Q is: LN (IN) *)</code>

2.18.6. POW ** POWL

Function Calculates a power

Inputs IN : REAL/LREAL Real value

EXP : REAL/LREAL Exponent

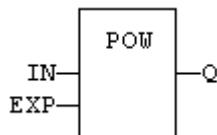
Outputs Q : REAL/LREAL Result: IN at the 'EXP' power

Remarks Alternatively, in ST language, the ****** operator can be used. In LD language, the operation is executed only if the input rung (EN) is *TRUE*. The output rung (ENO) keeps the same value as the input rung. In IL, the input must be loaded in the current result before calling the function. The exponent (second input of the function) must be the operand of the function.

ST Language `Q := POW (IN, EXP);`

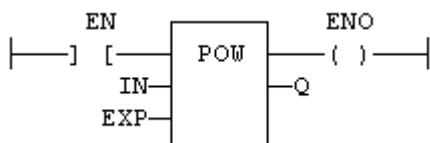
`Q := IN ** EXP;`

FBD Language



LD Language The function is executed only if EN is *TRUE*.

ENO keeps the same value as EN.



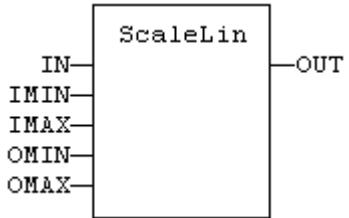
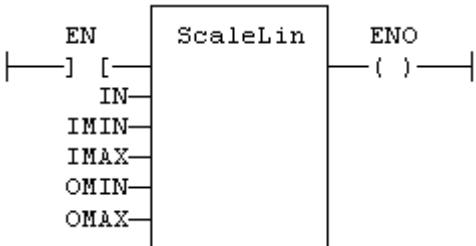
IL Language

Op1: LD IN

POW EXP

ST Q (* Q is: (IN ** EXP) *)

2.18.7. ScaleLin

Operator	Scaling - linear conversion	
Inputs	IN : REAL	Real value
	IMIN : REAL	Minimum input value
	IMAX : REAL	Maximum input value
	OMIN : REAL	Minimum output value
	OMAX : REAL	Maximum output value
Outputs	OUT : REAL	Result: OMIN + IN * (OMAX - OMIN) / (IMAX - IMIN)
Truth table	Inputs	OUT
	IMIN >= IMAX	= IN
	IN < IMIN	= IMIN
	IN > IMAX	= IMAX
	other	= OMIN + IN * (OMAX - OMIN) / (IMAX - IMIN)
Remarks	In LD language, the operation is executed only if the input rung (EN) is <i>TRUE</i> . The output rung (ENO) keeps the same value as the input rung. In IL, the input must be loaded in the current result before calling the function.	
ST Language	<code>OUT := ScaleLin (IN, IMIN, IMAX, OMIN, OMAX);</code>	
FBD Language		
LD Language	The function is executed only if EN is <i>TRUE</i> . ENO keeps the same value as EN.	
		
IL Language	<code>Op1: LD IN ScaleLin IMAX, IMIN, OMAX, OMIN ST OUT</code>	

2.18.8. SQRT / SQRTL

Function	Calculates the square root of the input
Inputs	IN : REAL/LREAL Real value
Outputs	Q : REAL/LREAL Result: square root of IN
Remarks	In LD language, the operation is executed only if the input rung (EN) is <i>TRUE</i> . The output rung (ENO) keeps the same value as the input rung. In IL, the input must be loaded in the current result before calling the function.
ST Language	<code>Q := SQRT (IN);</code>
FBD Language	
LD Language	The function is executed only if EN is <i>TRUE</i> . ENO keeps the same value as EN.
IL Language	<code>Op1: LD IN</code> <code>SQRT</code> <code>ST Q (* Q is: SQRT (IN) *)</code>

2.18.9. TRUNC / TRUNCL

Function Truncates the decimal part of the input

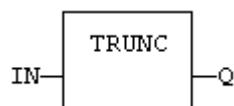
Inputs IN : REAL/LREAL Real value

Outputs Q : REAL/LREAL Result: integer part of IN

Remarks In LD language, the operation is executed only if the input rung (EN) is *TRUE*. The output rung (ENO) keeps the same value as the input rung. In IL, the input must be loaded in the current result before calling the function.

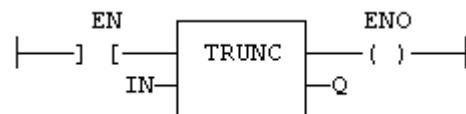
ST Language `Q := TRUNC (IN);`

FBD Language



LD Language The function is executed only if EN is *TRUE*.

ENO keeps the same value as EN.



IL Language `Op1: LD IN`

`TRUNC`

`ST Q (* Q is the integer part of IN *)`

2.19. Trigonometric functions

Below are the standard functions for trigonometric calculation:

Funktion	Effect
SIN	sine
COS	cosine
TAN	tangent
ASIN	arc-sine
ACOS	arc-cosine
ATAN	arc-tangent
ATAN2	arc-tangent of Y / X

2.19.1. ACOS /ACOSL

Function Calculate an arc-cosine

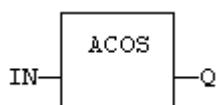
Inputs IN : REAL/LREAL Real value

Outputs Q : REAL/LREAL Result: arc-cosine of IN

Remarks In LD language, the operation is executed only if the input rung (EN) is *TRUE*. The output rung (ENO) keeps the same value as the input rung. In IL, the input must be loaded in the current result before calling the function.

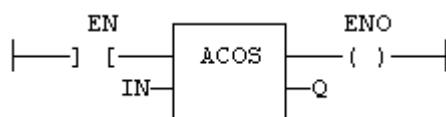
ST Language `Q := ACOS (IN);`

FBD Language



LD Language The function is executed only if EN is *TRUE*.

ENO keeps the same value as EN.



IL Language `Op1: LD IN`

`ACOS`

`ST Q (* Q is: ACOS (IN) *)`

2.19.2. ASIN / ASINL

Function Calculate an arc-sine

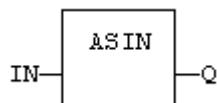
Inputs IN : REAL/LREAL Real value

Outputs Q : REAL/LREAL Result: arc-sine of IN

Remarks In LD language, the operation is executed only if the input rung (EN) is *TRUE*. The output rung (ENO) keeps the same value as the input rung. In IL, the input must be loaded in the current result before calling the function.

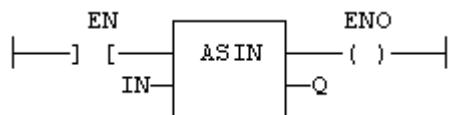
ST Language `Q := ASIN (IN);`

FBD Language



LD Language The function is executed only if EN is *TRUE*.

ENO keeps the same value as EN.



IL Language Op1: LD IN

ASIN

ST Q (* Q is: ASIN (IN) *)

2.19.3. ATAN / ATANL

Function Calculate an arc-tangent

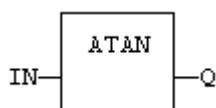
Inputs IN : REAL/LREAL Real value

Outputs Q : REAL/LREAL Result: arc-tangent of IN

Remarks In LD language, the operation is executed only if the input rung (EN) is *TRUE*. The output rung (ENO) keeps the same value as the input rung. In IL, the input must be loaded in the current result before calling the function.

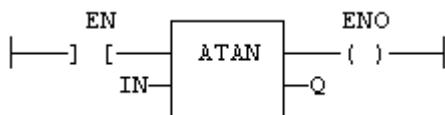
ST Language `Q := ATAN (IN);`

FBD Language



LD Language The function is executed only if EN is *TRUE*.

ENO keeps the same value as EN.



IL Language Op1: LD IN

ATAN

ST Q (* Q is: ATAN (IN) *)

2.19.4. ATAN2 / ATANL2

Function Calculate arc-tangent of Y/X

Inputs Y : REAL/LREAL Real value

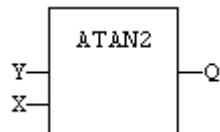
X : REAL/LREAL Real value

Outputs Q : REAL/LREAL Result: arc-tangent of Y / X

Remarks In LD language, the operation is executed only if the input rung (EN) is *TRUE*. The output rung (ENO) keeps the same value as the input rung. In IL, the input must be loaded in the current result before calling the function.

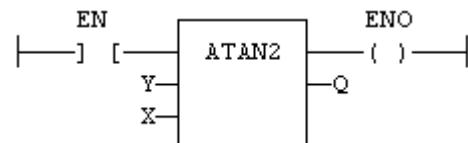
ST Language `Q := ATAN2 (IN);`

FBD Language



LD Language The function is executed only if EN is *TRUE*.

ENO keeps the same value as EN.



IL Language `Op1: LD Y`

`ATAN2 X`

`ST Q (* Q is: ATAN2 (Y / X) *)`

2.19.5. COS / COSL

Function Calculate a cosine

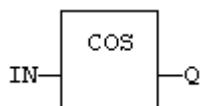
Inputs IN : REAL/LREAL Real value

Outputs Q : REAL/LREAL Result: cosine of IN

Remarks In LD language, the operation is executed only if the input rung (EN) is *TRUE*. The output rung (ENO) keeps the same value as the input rung. In IL, the input must be loaded in the current result before calling the function.

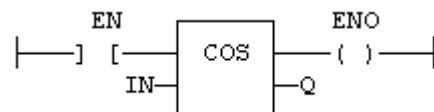
ST Language `Q := COS (IN);`

FBD Language



LD Language The function is executed only if EN is *TRUE*.

ENO keeps the same value as EN.



IL Language

`Op1: LD IN`

`COS`

`ST Q (* Q is: COS (IN) *)`



2.19.6. SIN / SINL

Function Calculate a sine

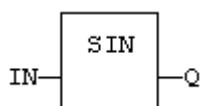
Inputs IN : REAL/LREAL Real value

Outputs Q : REAL/LREAL Result: sine of IN

Remarks In LD language, the operation is executed only if the input rung (EN) is *TRUE*. The output rung (ENO) keeps the same value as the input rung. In IL, the input must be loaded in the current result before calling the function.

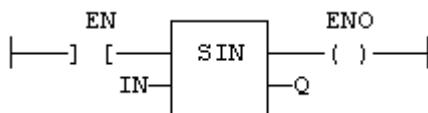
ST Language `Q := SIN (IN);`

FBD Language



LD Language The function is executed only if EN is *TRUE*.

ENO keeps the same value as EN.



IL Language Op1: LD IN

SIN

ST Q (* Q is: SIN (IN) *)



2.19.7. TAN / TANL

Function Calculate a tangent

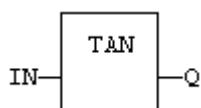
Inputs IN : REAL/LREAL Real value

Outputs Q : REAL/LREAL Result: tangent of IN

Remarks In LD language, the operation is executed only if the input rung (EN) is *TRUE*. The output rung (ENO) keeps the same value as the input rung. In IL, the input must be loaded in the current result before calling the function.

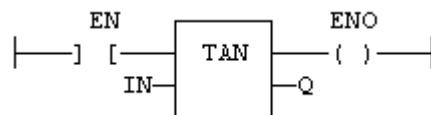
ST Language `Q := TAN (IN);`

FBD Language



LD Language The function is executed only if EN is *TRUE*.

ENO keeps the same value as EN.



IL Language

Op1: LD IN

TAN

ST Q (* Q is: TAN (IN) *)



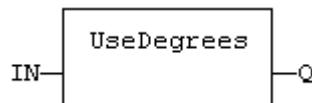
2.19.8. UseDegrees

Function	Sets the unit for angles in all trigonometric functions	
Inputs	IN : BOOL	If <i>TRUE</i> , turn all trigonometric functions to use degrees. If <i>FALSE</i> , turn all trigonometric functions to use radians (default)
Outputs	Q : BOOL	<i>TRUE</i> if functions use degrees before the call
Remarks	This function sets the working unit for the following functions:	

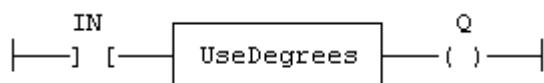
Code	Function
SIN	sine
COS	cosine
TAN	tangent
ASIN	arc-sine
ACOS	arc-cosine
ATAN	arc-tangent
ATAN2	arc-tangent of Y / X

ST Language Q := UseDegrees (IN);

FBD Language



LD Language Input is the rung. The rung is the output.



IL Language Op1: LD IN

 UseDegrees

 ST Q



2.20. String operations

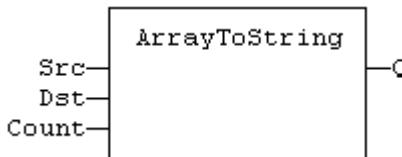
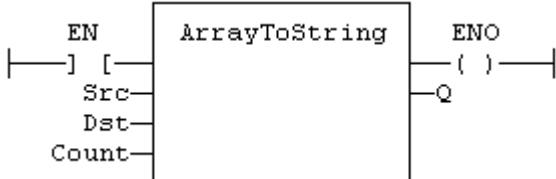
Below are the standard operators and functions that manage character strings:

Code	Operator / Function
+	concatenation of strings
CONCAT	concatenation of strings
MLEN	get string length
DELETE	delete characters in a string
INSERT	insert characters in a string
FIND	find characters in a string
REPLACE	replace characters in a string
LEFT	extract a part of a string on the left
RIGHT	extract a part of a string on the right
MID	extract a part of a string
CHAR	build a single character string
ASCII	get the ASCII code of a character within a string
ATOH	converts string to integer using hexadecimal basis
HTOA	converts integer to string using hexadecimal basis
CRC16	CRC16 calculation
ArrayToString	copies elements of an SINT array to a STRING
StringToArray	copies characters of a STRING to an SINT array

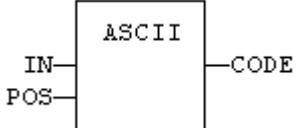
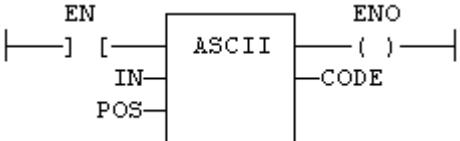
Other functions are available for managing string tables as resources:

Function	Description
StringTable	Select the active string table resource
LoadString	Load a string from the active string table

2.20.1. ArrayToString / ArrayToStringU

Function	Copy an array of SINT to a STRING
Inputs	SRC : SINT 6+ Source array of SINT small integers (USINT for ArrayToStringU) DST : STRING Destination STRING COUNT : DINT Numbers of characters to be copied
Outputs	Q : DINT Number of characters copied
Remarks	In LD language, the operation is executed only if the input rung (EN) is <i>TRUE</i> . The output rung (ENO) keeps the same value as the input rung. This function copies the COUNT first elements of the SRC array to the characters of the DST string. The function checks the maximum size of the destination string and adjust the COUNT number if necessary.
ST Language	<code>Q := ArrayToString (SRC, DST, COUNT);</code>
FBD Language	
LD Language	The function is executed only if EN is <i>TRUE</i> ENO keeps the same value as EN 
IL Language	Not available.

2.20.2. ASCII

Function	Get the ASCII code of a character within a string
Inputs	IN : STRING Input string POS : DINT Position of the character within the string (The first valid position is 1)
Outputs	CODE : DINT ASCII code of the selected character. or 0 if position is invalid
Remarks	In LD language, the input rung (EN) enables the operation, and the output rung keeps the same value as the input rung. In IL language, the first parameter (IN) must be loaded in the current result before calling the function. The other input is the operand of the function
ST Language	<code>CODE := ASCII (IN, POS);</code>
FBD Language	
LD Language	The function is executed only if EN is <i>TRUE</i> ENO is equal to EN 
IL Language	<code>Op1: LD IN AND_MASK MSK ST CODE</code>



2.20.3. ATOH

Function Converts string to integer using hexadecimal basis

Inputs IN : STRING String representing an integer in hexadecimal format

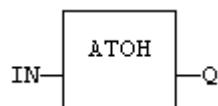
Outputs Q : DINT Integer represented by the string

IN	Q
"	0
'12'	18
'a0'	160
A0zzz'	160

Remarks The function is case insensitive. The result is 0 for an empty string. The conversion stops before the first invalid character. In LD language, the operation is executed only if the input rung (EN) is *TRUE*. The output rung (ENO) keeps the same value as the input rung. In IL, the input must be loaded in the current result before calling the function.

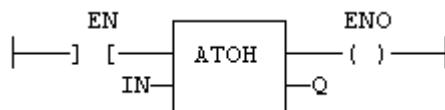
ST Language Q := ATOH (IN);

FBD Language



LD Language The function is executed only if EN is *TRUE*

ENO keeps the same value as EN

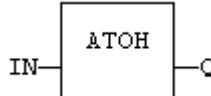
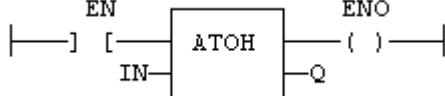
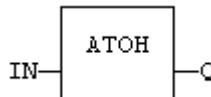
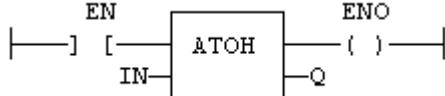


IL Language Op1: LD IN

ATOH

ST Q

2.20.4. CHAR

Function	Builds a single character string
Inputs	CODE : DINT ASCII code of the desired character
Outputs	Q : STRING STRING containing only the specified character
Remarks	In LD language, the input rung (EN) enables the operation, and the output rung keeps the same value as the input rung. In IL language, the input parameter (CODE) must be loaded in the current result before calling the function.
ST Language	Q := CHAR (CODE);
FBD Language	
LD Language	The function is executed only if EN is <i>TRUE</i> ENO keeps the same value as EN 
IL Language	Op1: LD CODE CHAR ST Q
Function	Builds a single character string
Inputs	CODE : DINT ASCII code of the desired character
Outputs	Q : STRING STRING containing only the specified character
Remarks	In LD language, the input rung (EN) enables the operation, and the output rung keeps the same value as the input rung. In IL language, the input parameter (CODE) must be loaded in the current result before calling the function.
ST Language	Q := CHAR (CODE);
FBD Language	
LD Language	The function is executed only if EN is <i>TRUE</i> ENO keeps the same value as EN 
IL Language	Op1: LD CODE



CHAR

ST Q

2.20.5. CONCAT

Function Concatenate strings

Inputs IN_1 : STRING Any string variable or constant expression

.....

IN_N : STRING Any string variable or constant expression.

Outputs Q : STRING Concatenation of all inputs

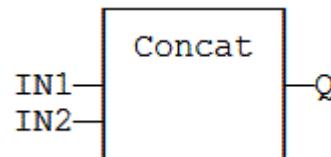
Remarks In FBD or LD language, the block may have up to 16 inputs. In IL or ST, the function accepts a variable number of inputs (at least 2).

Note that you also can use the "+" operator to concatenate strings.

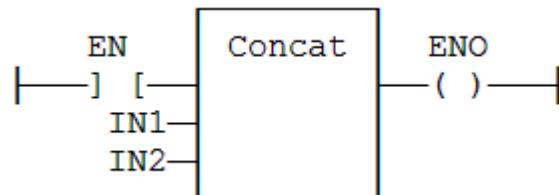
ST Language `Q := CONCAT ('AB', 'CD', 'E');`

(* now Q is 'ABCDE' *)

FBD Language



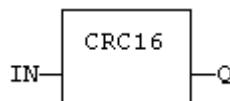
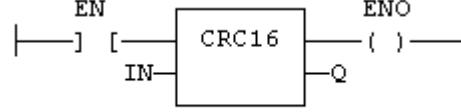
LD Language



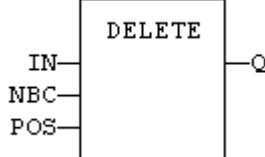
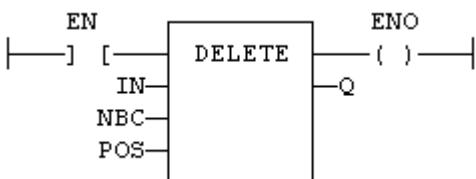
IL Language

`Op1: LD 'AB'`
`CONCAT 'CD', 'E'`
`ST Q (* Q is now 'ABCDE' *)`

2.20.6. CRC16

Function	Calculates a CRC16 on the characters of a string
Inputs	IN : STRING character string
Outputs	Q : INT CRC16 calculated on all the characters of the string
Remarks	In LD language, the input rung (EN) enables the operation, and the output rung keeps the same value as the input rung. In IL language, the input parameter (IN) must be loaded in the current result before calling the function. The function calculates a MODBUS CRC16, initialized at 16#FFFF value.
ST Language	<code>Q := CRC16 (IN);</code>
FBD Language	
LD Language	The function is executed only if EN is <i>TRUE</i> ENO is equal to EN 
IL Language	<code>Op1: LD IN CRC16 ST Q</code>

2.20.7. DELETE

Function	Delete characters in a string
Inputs	IN : STRING Character string NBC : DINT Number of characters to be deleted POS : DINT Position of the first deleted character (first character position is 1)
Outputs	Q : STRING Modified string
Remarks	The first valid character position is 1. In LD language, the operation is executed only if the input rung (EN) is <i>TRUE</i> . The output rung (ENO) keeps the same value as the input rung. In IL, the first input (the string) must be loaded in the current result before calling the function. Other arguments are operands of the function, separated by commas.
ST Language	<code>Q := DELETE (IN, NBC, POS);</code>
FBD Language	
LD Language	The function is executed only if EN is <i>TRUE</i> ENO keeps the same value as EN 
IL Language	<code>Op1: LD IN DELETE NBC, POS ST Q</code>



2.20.8. FIND

Function	Find position of characters in a string
Inputs	IN : STRING Character string STR : STRING String containing searched characters
Outputs	POS : DINT Position of the first character of STR in IN, or 0 if not found
Remarks	The first valid character position is 1. A return value of 0 means that the STR string has not been found. Search is case sensitive. In LD language, the operation is executed only if the input rung (EN) is <i>TRUE</i> . The output rung (ENO) keeps the same value as the input rung. In IL, the first input (the string) must be loaded in the current result before calling the function. The second argument is the operand of the function.
ST Language	<code>POS := FIND (IN, STR);</code>
FBD Language	
LD Language	The function is executed only if EN is <i>TRUE</i> ENO keeps the same value as EN
IL Language	<code>Op1: LD IN FIND STR ST POS</code>



2.20.9. HTOA

Function Converts integer to string using hexadecimal basis

Inputs IN : DINT Integer value

Outputs Q : STRING String representing the integer in hexadecimal format

Truth table

(Examples)

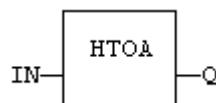
IN	Q
0	'0'
18	'12'
160	'A0'

Remarks

In LD language, the operation is executed only if the input rung (EN) is *TRUE*. The output rung (ENO) keeps the same value as the input rung. In IL, the input must be loaded in the current result before calling the function.

ST Language `Q := HTOA (IN);`

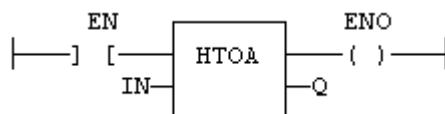
FBD Language



LD Language

The function is executed only if EN is *TRUE*

ENO keeps the same value as EN



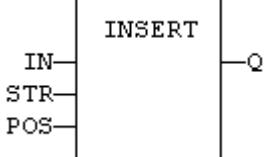
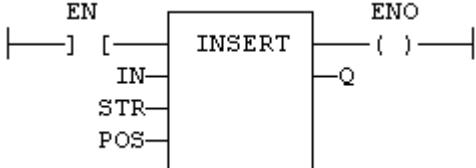
IL Language

Op1: LD IN

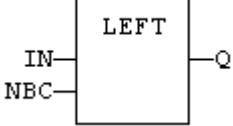
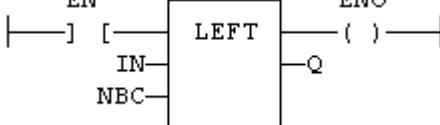
HTOA

ST Q

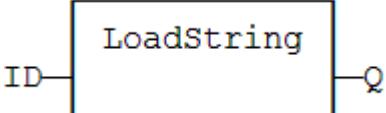
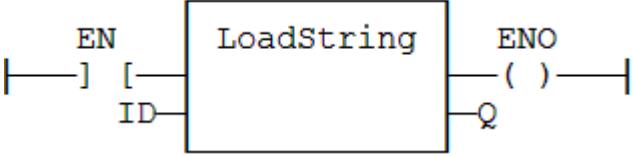
2.20.10. INSERT

Function	Insert characters in a string
Inputs	IN : STRING Character string STR : STRING String containing characters to be inserted POS : DINT Position of the first inserted character (first character position is 1)
Outputs	Q : STRING Modified string
Remarks	The first valid character position is 1. In LD language, the operation is executed only if the input rung (EN) is <i>TRUE</i> . The output rung (ENO) keeps the same value as the input rung. In IL, the first input (the string) must be loaded in the current result before calling the function. Other arguments are operands of the function, separated by commas.
ST Language	<code>Q := INSERT (IN, STR, POS);</code>
FBD Language	
LD Language	The function is executed only if EN is <i>TRUE</i> ENO keeps the same value as EN 
IL Language	<code>Op1: LD IN INSERT STR, POS ST Q</code>

2.20.11. LEFT

Function	Extract characters of a string on the left
Inputs	IN : STRING Character string NBC : DINT Number of characters to extract
Outputs	Q : STRING String containing the first NBC characters of IN
Remarks	In LD language, the operation is executed only if the input rung (EN) is <i>TRUE</i> . The output rung (ENO) keeps the same value as the input rung. In IL, the first input (the string) must be loaded in the current result before calling the function. The second argument is the operand of the function.
ST Language	<code>Q := LEFT (IN, NBC);</code>
FBD Language	
LD Language	The function is executed only if EN is <i>TRUE</i> ENO keeps the same value as EN 
IL Language	<code>Op1: LD IN LEFT NBC ST Q</code>

2.20.12. LoadString

Function	Load a string from the active string table
Inputs	ID : DINT ID of the string as declared in the string table
Outputs	Q : STRING Loaded string or empty string in case of error
Remarks	<p>This function loads a string from the active string table and stores it into a STRING variable. The StringTablefunction is used for selecting the active string table.</p> <p>The <i>ID</i> input (the string item identifier) is an identifier such as declared within the string table resource. You don't need to "define" this identifier again. The system will automatically define the identifier.</p>
ST Language	<pre>Q := LoadString (ID);</pre>
FBD Language	
LD Language	
IL Language	<pre>Op1: LD ID LoadString ST Q</pre>

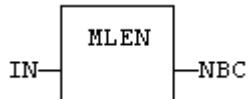
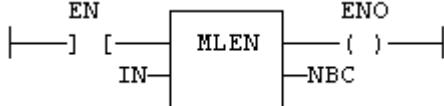


2.20.13. MID

Function	Extract characters of a string at any position
Inputs	IN : STRING Character string. NBC : DINT Number of characters to extract. POS : DINT Position of the first character to extract (first character of IN is at position 1).
Outputs	Q : STRING String containing the first NBC characters of IN
Remarks	The first valid position is 1. In LD language, the operation is executed only if the input rung (EN) is <i>TRUE</i> . The output rung (ENO) keeps the same value as the input rung. In IL, the first input (the string) must be loaded in the current result before calling the function. Other arguments are operands of the function, separated by commas.
ST Language	<code>Q := MID (IN, NBC, POS);</code>
FBD Language	
LD Language	The function is executed only if EN is <i>TRUE</i> . ENO keeps the same value as EN.
IL Language	<code>Op1: LD IN MID NBC, POS ST Q</code>

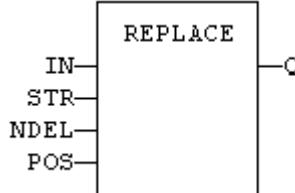
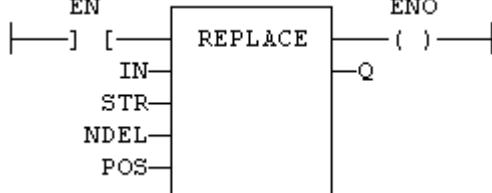


2.20.14. MLEN

Function	Get the number of characters in a string
Inputs	IN : STRING Character string
Outputs	NBC : DINT Number of characters currently in the string. 0 if string is empty
Remarks	In LD language, the operation is executed only if the input rung (EN) is <i>TRUE</i> . The output rung (ENO) keeps the same value as the input rung. In IL, the input must be loaded in the current result before calling the function.
ST Language	<code>NBC := MLEN (IN);</code>
FBD Language	
LD Language	The function is executed only if EN is <i>TRUE</i> . ENO keeps the same value as EN. 
IL Language	<code>Op1: LD IN</code> <code>MLEN</code> <code>ST NBC</code>



2.20.15. REPLACE

Function	Replace characters in a string								
Inputs	<table border="0"><tr><td>IN : STRING</td><td>Character string.</td></tr><tr><td>STR : STRING</td><td>String containing the characters to be inserted in place of NDEL removed characters</td></tr><tr><td>NDEL : DINT</td><td>Number of characters to be deleted before insertion of STR</td></tr><tr><td>POS : DINT</td><td>Position where characters are replaced (first character position is 1)</td></tr></table>	IN : STRING	Character string.	STR : STRING	String containing the characters to be inserted in place of NDEL removed characters	NDEL : DINT	Number of characters to be deleted before insertion of STR	POS : DINT	Position where characters are replaced (first character position is 1)
IN : STRING	Character string.								
STR : STRING	String containing the characters to be inserted in place of NDEL removed characters								
NDEL : DINT	Number of characters to be deleted before insertion of STR								
POS : DINT	Position where characters are replaced (first character position is 1)								
Outputs	Q : STRING Modified string								
Remarks	The first valid character position is 1. In LD language, the operation is executed only if the input rung (EN) is <i>TRUE</i> . The output rung (ENO) keeps the same value as the input rung. In IL, the first input (the string) must be loaded in the current result before calling the function. Other arguments are operands of the function, separated by commas.								
ST Language	<code>Q := REPLACE (IN, STR, NDEL, POS);</code>								
FBD Language									
LD Language	<p>The function is executed only if EN is <i>TRUE</i>. ENO keeps the same value as EN.</p> 								
IL Language	<p>Op1: LD IN REPLACE STR, NDEL, POS ST Q</p>								

2.20.16. RIGHT

Function Extract characters of a string on the right

Inputs IN : STRING Character string

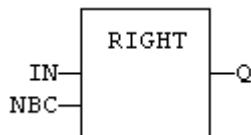
NBC : DINT Number of characters to extract

Outputs Q : STRING String containing the last NBC characters of IN

Remarks In LD language, the operation is executed only if the input rung (EN) is *TRUE*. The output rung (ENO) keeps the same value as the input rung. In IL, the first input (the string) must be loaded in the current result before calling the function. The second argument is the operand of the function.

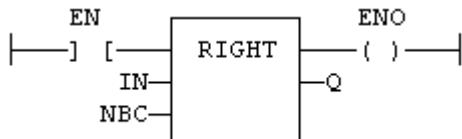
ST Language **Q := RIGHT (IN, NBC);**

FBD Language



LD Language The function is executed only if EN is *TRUE*.

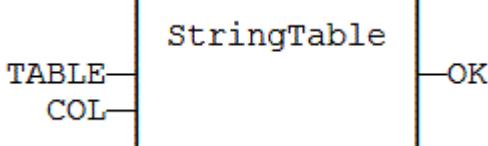
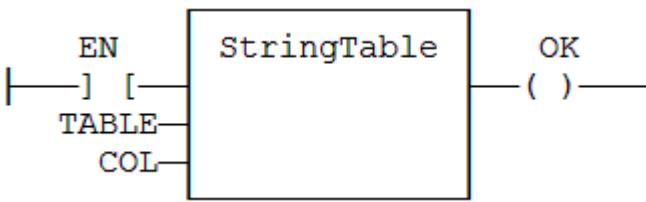
ENO keeps the same value as EN.



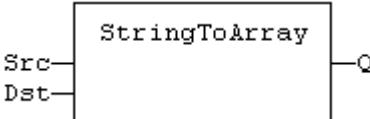
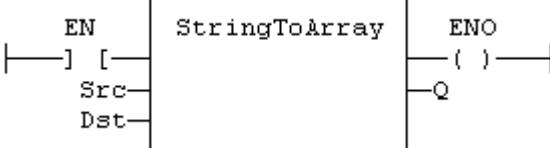
IL Language

Op1: LD IN
RIGHT NBC
ST Q

2.20.17. StringTable

Function	Selects the active string table
Inputs	TABLE : STRING Name of the String Table resource - must be a constant COL : STRING Name of the column in the table - must be a constant
Outputs	OK : BOOL <i>TRUE</i> if OK
Remarks	This function selects a column of a valid String Table resource to become the active string table. The LoadStringfunction always refers to the active string table. Arguments must be constant string expressions and must fit to a declared string table and a valid column name within this table. If you have only one string table with only one column defined in your project, you do not need to call this function as it will be the default string table anyway.
ST Language	<pre>OK := StringTable ('MyTable', 'FirstColumn');</pre>
FBD Language	
LD Language	
IL Language	<pre>Op1: LD 'MyTable' StringTable 'First Column' ST OK</pre>

2.20.18. StringToArray / StringToArrayU

Function	Copies the characters of a STRING to an array of SINT.
Inputs	SRC : STRING Source STRING. DST : SINT Destination array of SINT small integers (USINT for StringToArrayU).
Outputs	Q : DINT Number of characters copied
Remarks	In LD language, the operation is executed only if the input rung (EN) is <i>TRUE</i> . The output rung (ENO) keeps the same value as the input rung. In IL, the input must be loaded in the current result before calling the function. This function copies the characters of the SRC string to the first characters of the DST array. The function checks the maximum size destination arrays and reduces the number of copied characters if necessary.
ST Language	<code>Q := StringToArray (SRC, DST);</code>
FBD Language	
LD Language	The function is executed only if EN is TRUE. ENO keeps the same value as EN. 
IL Language	Op1: LD SRC StringToArray DST ST Q



2.21. Advanced operations

Below are the standard blocks that perform advanced operations.

2.21.1. Analog signal processing

Block	Description
Average	Calculate the average of signal samples
Integral	Calculate the integral of a signal
Derivate	Derive a signal
PID	PID loop
Ramp	Ramp signal
Lim_Alrm	Low / High level detection
Hyster	Hysteresis calculation
SigPlay	Play an analog signal from a resource
SigScale	Get a point from a signal resource
CurveLin	Linear interpolation on a curve
SurfLin	Linear interpolation on a surface

2.21.2. Alarm management

Block	Description
Lim_Alrm	Low / High level detection
Alarm_M	Alarm with manual reset
Alarm_A	Alarm with automatic reset



2.21.3. Data collections and serialization

Block	Description
StackInt	Stack of integers
FIFO	"First in / first out" list
LIFO	"Last in / first out" stack
SerializeIn	Extract data from a binary frame
SerializeOut	Write data to a binary frame
SerGetString	Extract a string from a binary frame
SerPutString	Copies a string to a binary frame

2.21.4. Data Logging

Block	Description
LogFileCSV	Log values of variables to a CSV file.

2.21.5. Special operations

Block	Description
GetSysInfo	Get system Information
Printf	Trace messages
CycleStop	Sets the application in cycle stepping mode
FatalStop	Breaks the cycle and stop with fatal error
EnableEvents	Enable / disable produced events for binding
ApplyRecipeColumn	Apply the values of a column from a recipe file
VLID	Get the ID of an embedded list of variables
SigID	Get the ID of a signal resource



2.21.6. Communication

SERIO	serial communication		
AS	interface		
TCP-IP	management functions		
UDP	management functions		
MQTT	protocol handling		
MBSlaveRTU	MBSlaveUDP	MBMasterRTU	MBMasterTCP
CanRcvMsg	CanSndMsg		
CANopen	functions		
DNP3	Master function blocks		

2.21.7. Others

File management functions	Dynamic memory allocation functions
Real Time Clock	Variable size text buffers manipulation
XML writing and parsing	T5 Registry management functions

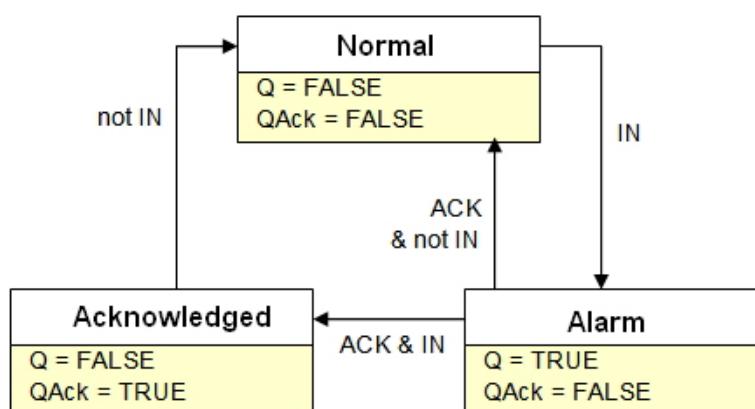
2.21.8. PTPALARMA

Function block Alarm with automatic reset

Inputs IN : BOOL Process signal
ACK : BOOL Acknowledge command

Outputs Q : BOOL TRUE if alarm is active
QACK : BOOL TRUE if alarm is acknowledged

Sequence

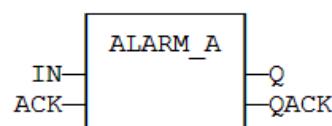


Remarks Combine this block with the LIM_ALRM block for managing analog alarms.

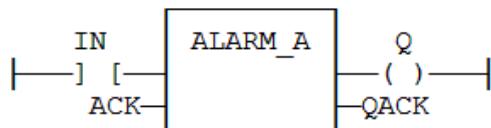
ST Language MyALARM is declared as an instance of ALARM_A function block.

```
MyALARM (IN, ACK, RST);  
Q := MyALARM.Q;  
QACK := MyALARM.QACK;
```

FBD Language



LD Language



IL Language MyALARM is declared as an instance of ALARM_A function block.

```
Op1: CAL MyALARM (IN, ACK, RST)  
LD MyALARM.Q  
ST Q  
LD MyALARM.QACK  
ST QACK
```

2.21.9. ALARM_M

Function block Alarm with manual reset

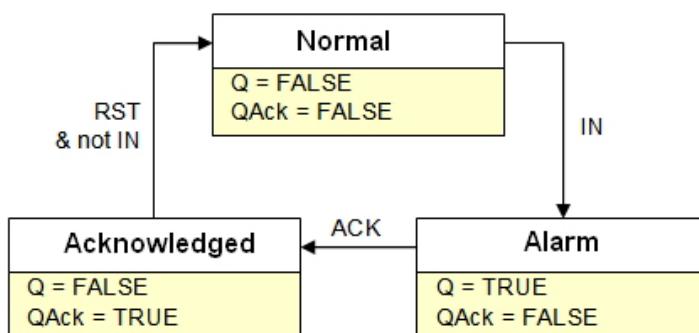
Inputs

IN : BOOL	Process signal
ACK : BOOL	Acknowledge command
RST : BOOL	Reset command

Outputs

Q : BOOL	TRUE if alarm is active
QACK : BOOL	TRUE if alarm is acknowledged

Sequence

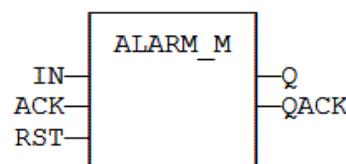


Remarks Combine this block with the LIM_ALRM block for managing analog alarms.

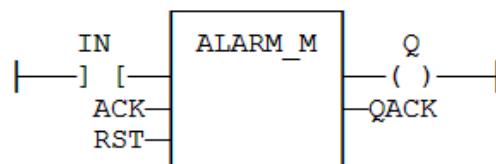
ST Language MyALARM is declared as an instance of ALARM_M function block.

```
MyALARM (IN, ACK, RST);  
Q := MyALARM.Q;  
QACK := MyALARM.QACK;
```

FBD Language



LD Language



IL Language MyALARM is declared as an instance of ALARM_M function block.

```
Op1: CAL MyALARM (IN, ACK, RST)  
LD MyALARM.Q  
ST Q  
LD MyALARM.QACK  
ST QACK
```

2.21.10. ApplyRecipeColumn

Function block Apply the values of a column from a recipe file

Inputs FILE : STRING Pathname of the recipe file (.RCP or .CSV) - must be a constant value!

COL : DINT Index of the column in the recipe (0 based)

Outputs OK : BOOL TRUE if OK - FALSE if parameters are invalid

Remarks The 'FILE' input is a constant string expression specifying the path name of a valid RCP or CSV file. If no path is specified, the file is assumed to be located in the project folder. RCP files are created using the recipe editor. CSV files can be created using EXCEL or NOTEPAD.

In CSV files, the first line must contain column headers, and is ignored during compiling. There is one variable per line. The first column contains the symbol of the variable. Other columns are values.

If a cell is empty, it is assumed to be the same value as the previous (left side) cell. If it is the first cell of a raw, it is assumed to be null (0 or FALSE or empty string).

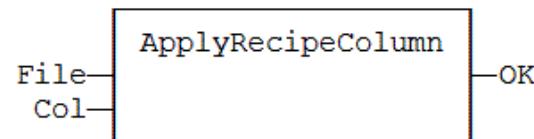
In LD language, the operation is executed only if the input rung (EN) is TRUE. The output rung is the result of the function.

Attention:

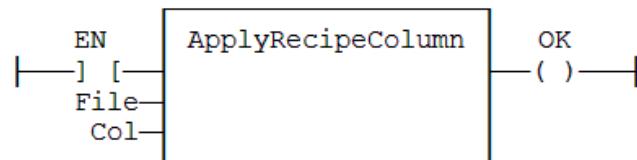
Recipe files are read at compiling time and are embedded into the downloaded application code. This implies that a modification performed in the recipe file after downloading will not be taken into account by the application.

ST Language `OK := ApplyRecipeColumn ('MyFile.rep', COL);`

FBD Language



LD Language The function is executed only if EN is TRUE.



IL Language Op1: LD 'MyFile.rcp'

 ApplyRecipeColumn COL

 ST OK



2.21.11. AS-interface functions

The following functions enable special operation on AS-i networks:

Block	Description, Interface, and Arguments
Master/Slave Arguments	Master : DINT Index of the AS-i master (1..N) such as shown in configuration Slave : DINT Address of the AS-i slave (1..32 / 33..63)
ASiReadPP	read permanent parameters of an AS-i slave Params := ASiReadPP (Master, Slave); Params : DINT Value of AS-i parameters.
ASiWritePP	write permanent parameters of an AS-i slave bOK := ASiWritePP (Master, Slave, Params); bOK : BOOL TRUE if successful
ASiSendParam	send parameters to an AS-i slave bOK := ASiSendParam (Master, Slave, Params); bOK : BOOL TRUE if successful
ASiReadPI	read actual parameters of an AS-i slave Params := ASiReadPI (Master, Slave); Params : DINT Value of AS-i parameters.
ASiStorePI	store actual parameters as permanent parameters bOK := ASiStorePI (Master); bOK : BOOL TRUE if successful
Attention: AS-i networking may be not available on some targets. Please refer to OEM instructions for further details about available features.	

2.21.12. AVERAGE / AVERAGEL

Function block Calculates the average of signal samples

Inputs RUN : BOOL Enabling command

XIN : REAL Input signal (*)

N : DINT Number of samples stored for average calculation - Cannot exceed 128

Outputs XOUT : REAL Average of the stored samples, where AVERAGEL has LREAL arguments

Remarks The average is calculated according to the number of stored samples, that can be less than N when the block is enabled. In LD language, the input rung is the RUN command. The output rung keeps the state of the input rung.

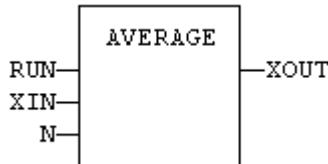
The "N" input is taken into account only when the RUN input is *FALSE*. So the "RUN" needs to be reset after a change.

ST Language MyAve is a declared instance of AVERAGE function block

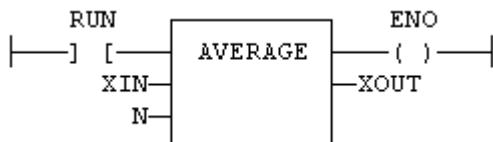
MyAve (RUN, XIN, N);

XOUT := MyAve.XOUT;

FBD Language



LD Language ENO has the same state as RUN.



IL Language MyAve is a declared instance of AVERAGE function block

Op1: CAL MyAve (RUN, XIN, N)

LD MyAve.XOUT

ST XOUT



2.22. CANopen functions

A set of functions and function blocks is available for management of CANopen devices. Such functions refer to the CANopen master configuration defined in the Fieldbus Configurator.

Sending NMT messages

The following functions are used to send a NMT message to a CANopen slave:

```
OK := coNmtReset (PORT, SLV); (* reset node *)
OK := coNmtStart (PORT, SLV); (* start node *)
OK := coNmtStop (PORT, SLV); (* stop node *)
OK := coNmtEnterPreop (PORT, SLV); (* enter peroperational *)
```

Arguments

OK : BOOL;(* TRUE if successful, FALSE if busy or slave not configured *)

PORT : STRING; (* Name of the CAN port as specified in the CANopen master configuration *)

SLV : DINT; (* CANopen slave number *)

Note:

You cannot sent more than one NMT message to the same slave during the same PLC cycle.

Reading a simple parameter (SDO) coReadParam function block

The coReadParam function block is used for sending a simple SDO message to read a parameter from a slave.

Input arguments

EN : BOOL; (* the message is sent on a rising edge of this input *)

PORT : STRING; (* Name of the CAN port as specified in the CANopen master configuration *)

SLV : DINT; (* CANopen slave number *)

Param : DINT; (* SDO parameter index *)

Sub : DINT; (* SDO parameter sub-index *)

TO : TIME; (* timeout for exchange *)

Output arguments

Ready : BOOL; (* TRUE when the block is ready - FALSE during an exchange *)

RC : DINT; (* CANopen return check *)

VAL : DINT; (* read parameter value answered by the slave *)



Writing a simple parameter (SDO) coWriteParam function block

The coWriteParam function block is used for sending a simple SDO message to write a parameter to a slave.

Input arguments	EN : BOOL; (* the message is sent on a rising edge of this input *) PORT : STRING; (* Name of the CAN port as specified in the CANopen master configuration *) SLV : DINT; (* CANopen slave number *) Param : DINT; (* SDO parameter index *) Sub : DINT; (* SDO parameter sub-index *) VAL : DINT; (* parameter value *) LEN : DINT; (* number of bytes to send - from 1 to 4 *) TO : TIME; (* timeout for exchange *)
Output arguments	Ready : BOOL; (* TRUE when the block is ready - FALSE during an exchange *) RC : DINT; (* CANopen return check *)

Sending the PDO mapping (SDOs) coSendPdoMap function block

The coSendPdoMap function block sends a sequence of SDO messages for applying the PDO mapping to a slave such as defined in the CANopen configuration.

Input arguments	EN : BOOL; (* the message is sent on a rising edge of this input *) PORT : STRING; (* Name of the CAN port as specified in the CANopen master configuration *) SLV : DINT; (* CANopen slave number *) TO : TIME; (* timeout for exchange *)
Output arguments	Ready : BOOL; (* TRUE when the block is ready - FALSE during the exchanges *) RC : DINT; (* CANopen return check *)



2.22.1. CanRcvMsg

Function block Receive a CANbus message

Inputs	PORT : STRING	CAN port identification (see remarks)
	DATA : USINT[]	CAN data (filled on output) - array size must be 8
Outputs	OK : BOOL	TRUE if successful
	ID : DWORD	CAN message ID
	LEN : DINT	CAN message data length

Remarks This function gets the next received message since the last cycle. Only messages with IDs not configured in the CANbus configuration are received by this block.

The number of messages possibly received within one cycle is limited by the FIFO size defined in the CANbus configuration.

The PORT input defines a CAN bus configured:

- If you specify an empty string, then the first CAN port configured is selected
- If you specify a string containing a number, it relates to the corresponding port. For instance, '2' means the second port
- You can also specify the string entered as "port settings" in the CANbus configuration, e.g. 'CAN0'

2.22.2. CanSndMsg

Function block Send a CANbus message

Inputs	PORT : STRING	CAN port identification (see remarks)
	ID : DWORD	CAN message ID
	LEN : DINT	CAN message data length
	DATA : USINT[]	CAN data
	RTR : BOOL	Send in RTR mode

Outputs OK : BOOL TRUE if successful

Remarks This function sends an asynchronous message on the specified CAN port. The message will be sent at the end of the current cycle.

The number of messages possibly sent within one cycle is limited by the FIFO size defined in the CANbus configuration.

The PORT input defines a CAN bus configured:

- If you specify an empty string, then the first CAN port configured is selected
- If you specify a string containing a number, it relates to the corresponding port. For instance, '2' means the second port
- You can also specify the string entered as "port settings" in the CANbus configuration, e.g. 'CAN0'



2.22.3. CycleStop

Function block Sets the application in cycle stepping mode

Inputs IN : BOOL Condition

Outputs Q : BOOL *TRUE* if performed

Remarks This function turns the Virtual Machine in *Cycle Stepping* mode. Restarting normal execution will be performed using the debugger.

The VM is set in *cycle stepping* mode only if the IN argument is *TRUE*.

The current main program and all possibly called sub-programs or UDFBs are normally performed up the end. Other programs of the cycle are not executed.

2.22.4. CurveLin

Function block Linear interpolation on a curve

Inputs X : REAL X coordinate of the point to be interpolated

XAxis : REAL[] X coordinates of the known points of the X axis

YVal : REAL[] Y coordinate of the points defined on the X axis

Outputs Y : REAL Interpolated Y value corresponding to the X input

OK : BOOL *TRUE* if successful

ERR : DINT Error code if failed - 0 if OK

Remarks This function performs linear interpolation in between a list of points defined in the XAxis single dimension array. The output Y value is an interpolation of the Y values of the two rounding points defined in the X axis. Y values of defined points are passed in the YVal single dimension array.

Values in XAxis must be sorted from the smallest to the biggest. There must be at least two points defined in the X axis. YVal and XAxis input arrays must have the same dimension.

In case the X input is less than the smallest defined X point, the Y output takes the first value defined in YVal and an error is reported. In case the X input is greater than the biggest defined X point, the Y output takes the last value defined in YVal and an error is reported.

The ERR output gives the cause of the error if the function fails:

Error code	Meaning
0	OK
1	Invalid dimension of input arrays
2	Invalid points for the X axis
4	X is out of the defined X axis

2.22.5. DERIVATE

Function block Derivates a signal

Inputs RUN : BOOL Run command: *TRUE*=derivate / *FALSE*=hold

XIN : REAL Input signal

CYCLE : TIME Sampling period (should not be less than the target cycle timing)

Outputs XOUT : REAL Output signal

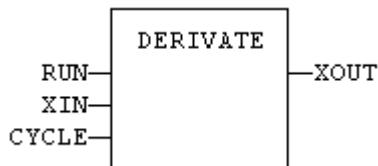
Remarks In LD language, the input rung is the RUN command. The output rung keeps the state of the input rung.

ST Language MyDerv is a declared instance of DERIVATE function block.

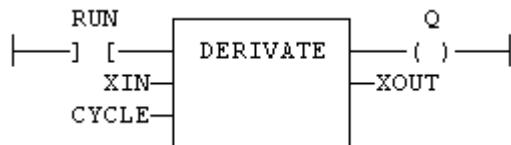
MyDerv (RUN, XIN, CYCLE);

XOUT := MyDerv.XOUT;

FBD Language



LD Language



IL Language MyDerv is a declared instance of DERIVATE function block.

Op1: CAL MyDerv (RUN, XIN, CYCLE)

LD MyDerv.XOUT

ST XOUT



2.23. DNP3 Master function blocks

2.23.1. DNP3_CMDANLG

Function block Issue an analog command

Inputs	EN : BOOL	Activate the block
	SESN : UINT	Session number
	PNUM : UINT	Point number
	FC : UINT	Fucntion code - must be DNP3_FC_SELECT or DNP3_FC_OPERATE or DNP3_FC_DIRECT_OP or DNP3_FC_DIRECT_OP_NOACK
	VART : UINT	Object variation to use in request
	QUAL : UINT	Qualifier - must be DNP3_QUAL_8BIT_INDEX or DNP3_QUAL_16BIT_INDEX
	VALUE : ANY	Value to write - can be any data type
Outputs	READY : BOOL	TRUE if the block is ready for a new command
	RC : DINT	Status: see values DNP3_RESP_STATUS_. at the end of this page



2.23.2. DNP3_CMDBIN

Function block Issue a binary command

Inputs	EN : BOOL	Activate the block
	SESN : UINT	Session number
	PNUM : UINT	Point number
	FC : UINT	Fucntion code - must be DNP3_FC_SELECT or DNP3_FC_OPERATE or DNP3_FC_DIRECT_OP or DNP3_FC_DIRECT_OP_NOACK
	CONTROL : UINT	Control code - must be DNP3_CROB_CTRL_PULSE_ON or DNP3_CROB_CTRL_PULSE_OFF or DNP3_CROB_CTRL_LATCH_ON or DNP3_CROB_CTRL_LATCH_OFF or DNP3_CROB_CTRL_CLEAR or DNP3_CROB_CTRL_PAIRED_CLOSE or DNP3_CROB_CTRL_PAIRED_TRIP
	QUAL : UINT	Qualifier - must be DNP3_QUAL_8BIT_INDEX or DNP3_QUAL_16BIT_INDEX
	TON : TIME	ON duration
	TOFF : TIME	OFF duration
Outputs	READY : BOOL	TRUE if the block is ready for a new command
	RC : DINT	Status: see values DNP3_RESP_STATUS_. at the end of this section



2.23.3. DNP3_EVPOLL

Function block Issue an event data poll

Inputs	EN : BOOL	Activate the block
	SESN : UINT	Session number
Outputs	READY : BOOL	TRUE if the block is ready for a new command
	RC : DINT	Status: see values DNP3_RESP_STATUS_. at the end of this section

2.23.4. DNP3_FREEZCNTR

Function block Issue a freeze counters request

Inputs	EN : BOOL	Activate the block
	SESN : UINT	Session number
	QUAL : UINT	Qualifier - must be DNP3_QUAL_ALL_POINTS or DNP3_QUAL_8BIT_START_STOP or DNP3_QUAL_16BIT_START_STOP See values at the end of this page
	START : UINT	Start point if qualifier is 8/16 bit start/stop
	STOP : UINT	Stop point if qualifier is 8/16 bit start/top
Outputs	READY : BOOL	TRUE if the block is ready for a new command
	RC : DINT	Status: see values DNP3_RESP_STATUS_. at the end of this section

2.23.5. DNP3_INTPOLL

Function block Issue an integrity data poll

Inputs	EN : BOOL	Activate the block
	SESN : UINT	Session number
Outputs	READY : BOOL	TRUE if the block is ready for a new command
	RC : DINT	Status: see values DNP3_RESP_STATUS_. at the end of this section



2.23.6. DNP3_READCLASS

Function block Issue a read class request

Inputs	EN : BOOL	Activate the block
	SESN : UINT	Session number
	QUAL : UINT	Qualifier - must be DNP3_QUAL_ALL_POINTS or DNP3_QUAL_8BIT_LIMITED_QTY or DNP3_QUAL_16BIT_LIMITED_QTY
	QTY : UINT	Quantity to request if qualifier is 8/16 bit limited quantity
	CLASS0 : BOOL	TRUE to request class 0 (static data)
	CLASS1 : BOOL	TRUE to request class 1 event data
	CLASS2 : BOOL	TRUE to request class 2 event data
	CLASS3 : BOOL	TRUE to request class 3 event data
Outputs	READY : BOOL	TRUE if the block is ready for a new command
	RC : DINT	Status: see values DNP3_RESP_STATUS_. at the end of this section



2.23.7. DNP3_READGROUP

Function block Issue a read group request to the specified session

Inputs	EN : BOOL	Activate the block
	SESN : UINT	Session number
	GRP : UINT	Object group for read request
	VART : UINT	object variation to read. The type specified by this is specific to the object group being read
	QUAL : UINT	Qualifier - must be DNP3_QUAL_ALL_POINTS or DNP3_QUAL_8BIT_START_STOP or DNP3_QUAL_16BIT_START_STOP or DNP3_QUAL_8BIT_LIMITED_QTY or DNP3_QUAL_16BIT_LIMITED_QTY
	START : UINT	Start point if qualifier is start/stop
	STOP : UINT	Stop point if qualifier is start/stop or quantity if qualifier is limited points
	EN : BOOL	Activate the block
Outputs	READY : BOOL	TRUE if the block is ready for a new command
	RC : DINT	Status: see values DNP3_RESP_STATUS_. at the end of this section

2.23.8. DNP3_READPTS

Function block Issue an read one or more specific data points request to the specified session

Inputs	EN : BOOL	Activate the block
	SESN : UINT	Session number
	GRP : UINT	Object group for read request
	VART : UINT	object variation to read
	QUAL : UINT	Qualifier - must be DNP3_QUAL_8BIT_INDEX or DNP3_QUAL_16BIT_INDEX
	ARPTS[] : UINT	Array of point numbers to read
Outputs	READY : BOOL	TRUE if the block is ready for a new command
	RC : DINT	Status: see values DNP3_RESP_STATUS_. at the end of this section



2.23.9. DNP3_WRITEBIN

Function block Issue a binary output write command to the specified session

Inputs	EN : BOOL	Activate the block
	SESN : UINT	Session number
	QUAL : UINT	Qualifier - must be DNP3_QUAL_8BIT_START_STOP or DNP3_QUAL_16BIT_START_STOP
	START : UINT	Start point index
	STOP : UINT	Endi point index
	VALUE[] : BOOL	Array of values (FALSE = off / TRUE = on)
Outputs	READY : BOOL	TRUE if the block is ready for a new command
	RC : DINT	Status: see values DNP3_RESP_STATUS_. at the end of this section

2.23.10. DNP3_WRITESTR

Function block Write a string object to the specified point

Inputs	EN : BOOL	Activate the block
	SESN : UINT	Session number
	PNUM : UINT	Point number for the string to write
	STR : STRING	String value
	STOP : UINT	Endi point index
	VALUE[] : BOOL	Array of values (FALSE = off / TRUE = on)
Outputs	READY : BOOL	TRUE if the block is ready for a new command
	RC : DINT	Status: see values DNP3_RESP_STATUS_. at the end of this section



2.23.11. Command and status values

Below are command values and status values used by or returned by DNP3 Master function blocks. You can directly copy / paste these definitions in the "Global Defines" editor.

Qualifier

DNP3_QUAL_8BIT_START_STOP	16#00
DNP3_QUAL_16BIT_START_STOP	16#01
DNP3_QUAL_ALL_POINTS	16#06
DNP3_QUAL_8BIT_LIMITED_QTY	16#07
DNP3_QUAL_16BIT_LIMITED_QTY	16#08
DNP3_QUAL_8BIT_INDEX	16#17
DNP3_QUAL_16BIT_INDEX	16#28

Function code

DNP3_FC_SELECT	16#03
DNP3_FC_OPERATE	16#04
DNP3_FC_DIRECT_OP	16#05
DNP3_FC_DIRECT_OP_NOACK	16#06

Values for control code of CROB

DNP3_CROB_CTRL_PULSE_ON	16#01	The point is turned on for the specified onTime, turned off for the specified offTime, and left in the off state
DNP3_CROB_CTRL_PULSE_OFF	16#02	The point is turned off for the specific offTime, then turned on for the specified onTime, and left in the on state
DNP3_CROB_CTRL_LATCH_ON	16#03	The point is latched on
DNP3_CROB_CTRL_LATCH_OFF	16#04	The point is latched off
DNP3_CROB_CTRL_CLEAR	16#20	Cancel the currently running operation
DNP3_CROB_CTRL_PAIED_CLOSE	16#40	Activate Close relay
DNP3_CROB_CTRL_PAIED_TRIP	16#80	Activate Trip relay



Response values

DNP3_RESP_STATUS_SUCCESS	16#00	This indicates the request has completed successfully
DNP3_RESP_STATUS_INTERMEDIATE	16#01	This indicates a response was received but the requested command is not yet complete
DNP3_RESP_STATUS_FAILURE	16#02	This indicates that the transmission of the request failed
DNP3_RESP_STATUS_MISMATCH	16#03	The response to a select or an execute did not echo the request
DNP3_RESP_STATUS_STATUSCODE	16#04	The response to a select or an execute echoed the request, except the status code was different indicating a failure
DNP3_RESP_STATUS_IIN	16#05	The response to the request had IIN bits set indicating the command failed
DNP3_RESP_STATUS_TIMEOUT	16#06	This indicates that the request has timed out
DNP3_RESP_STATUS_CANCELED	16#07	Request canceled by user
DNP3_RESP_STATUS_OTHERERR	16#ff	Command failed (ie. Variation not supported)



2.24. Dynamic memory allocation functions

The following functions enable the dynamic allocation of arrays for storing DINT integer values:

ARCREATE	allocates an array of DINT integers
ARREAD	read a DINT integer in an array allocated by ARCREATE
ARWRITE	write a DINT integer in an array allocated by ARCREATE

Notes:

- The memory used for those arrays is allocated directly by the Operating System of the target. There is no insurance that the required memory space will be available.
- Allocating large arrays may cause the Operating System to be instable or slow down the performances of the target system.
- Dynamic memory allocation may be unsuccessful (not enough memory available on the target). Your application should process such error cases in a safe way.
- Dynamic memory allocation may be not available on some targets. Please refer to OEM instructions for further details about available features.



2.24.1. ARCREATE

Function	Array allocation
Command	OK := ARCREATE (ID, SIZE);
Inputs	ID : DINT integer ID to be assigned to the array (first possible ID is 0) SIZE : DINT desired number of DINT values to be stored in the array OK : DINT return check
Return values	1. OK - array is allocated and ready for read / write operations. 2. The specified ID is invalid or already used for another array. 3. The specified size is invalid. 4. Not enough memory (action denied by the Operating System). 5. The memory allocated by ARCREATE will be released when the application stops.

2.24.2. ARREAD

Function	Read array element
Command	VAL := ARREAD (ID, POS);
Inputs	ID : DINT integer ID of the array POS : DINT index of the element in the array (first valid index is 0) VAL : DINT value of the specified item or 0 if arguments are invalid

2.24.3. ARWRITE

Function	Write array element
Command	OK := ARWRITE (ID, POS, IN);
Inputs	ID : DINT integer ID of the array POS : DINT index of the element in the array (first valid index is 0) IN : DINT value to be assigned to the element OK : DINT return check
Return values	1. OK - element was forced successfully. 2. The specified ID is invalid (not an allocated array). 3. The specified index is invalid (out of array bounds).

2.24.4. EnableEvents

Function Enable or disable the production of events for binding (runtime to runtime variable exchange)

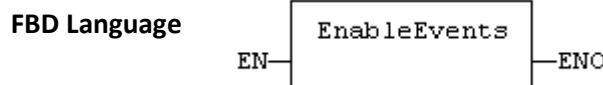
Inputs EN : BOOL *TRUE* to enable events / *FALSE* to disable events

Outputs ENO : BOOL Echo of EN input

Remarks Production is enabled when the application starts. The first production will be operated after the first cycle. To disable events from the beginning call *EnableEvents (FALSE)* in the very first cycle.

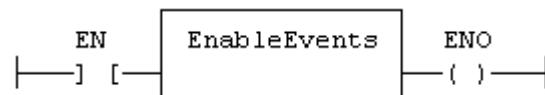
In LD language, the input rung (EN) enables the event production, and the output rung keeps the state of the input rung. In IL language, the input must be loaded before the function call.

ST Language ENO := EnableEvents (EN);



LD Language Events are enables if EN is *TRUE*.

ENO has the same value as EN.



IL Language Op1: LD EN

 EnableEvents

 ST ENO

2.24.5. FatalStop

Function Manages a first in / first out list

Inputs IN : BOOL Condition

Outputs Q : BOOL *TRUE* if performed

Remarks This function breaks the current cycle and sets the Virtual Machine in *ERROR* mode. Restarting normal execution will be performed using the debugger.

The VM is stopped only if the IN argument is *TRUE*. The end of the current cycle is then not performed.



2.24.6. FIFO

Function Enable or disable the production of events for binding (runtime to runtime variable exchange)

Inputs

- PUSH : BOOL Push a new value (on rising edge)
- POP : BOOL Pop a new value (on rising edge)
- RST : BOOL Reset the list
- NEXTIN : ANY Value to be pushed
- NEXTOUT : ANY Value of the oldest pushed value - updated after call!
- BUFFER : ANY Array for storing values

Outputs

- EMPTY : BOOL TRUE if the list is empty
- OFLO : BOOL TRUE if overflow on a PUSH command
- COUNT : DINT Number of values in the list
- PREAD : DINT Index in the buffer of the oldest pushed value
- PWRITE : DINT Index in the buffer of the next push position

Remarks NEXTIN, NEXTOUT and BUFFER must have the same data type and cannot be STRING.

The NEXTOUT argument specifies a variable that is filled with the oldest push value after the block is called.

Values are stored in the *BUFFER* array. Data is arranged as a roll over buffer and is never shifted or reset. Only read and write pointers and pushed values are updated. The maximum size of the list is the dimension of the array.

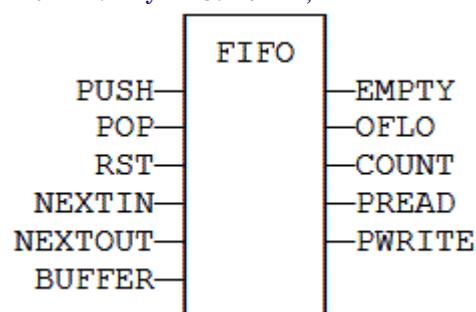
The first time the block is called, it remembers on which array it should work. If you call later the same instance with another BUFFER input, the call is considered as invalid and makes nothing. Outputs reports an empty list in this case.

In LD language, input rung is the PUSH input. The output rung is the EMPTY output.

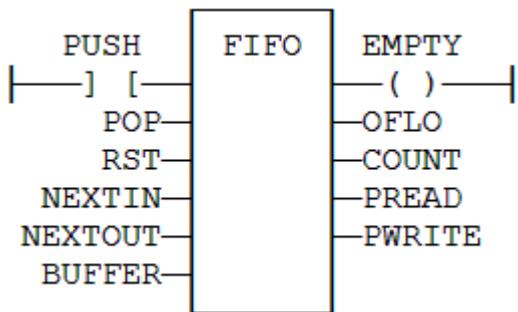
ST Language MyFIFO is a declared instance of FIFO function block.

```
SMyFIFO (PUSH, POP, RST, NEXTIN, NEXTOUT, BUFFER );  
EMPTY := MyFIFO.EMPTY;  
OFLO := MyFIFO.OFLO;  
COUNT := MyFIFO.COUNT;  
PREAD := MyFIFO.PREAD;
```

FBD Language



LD Language



IL Language

MyFIFO is a declared instance of FIFO function block.

```
Op1: CAL MyFIFO (PUSH, POP, RST, NEXTIN, NEXTOUT, BUFFER )
      LD MyFIFO.EMPTY
      ST EMPTY
      LD MyFIFO.OFLO
      ST OFLO
      LD MyFIFO.COUNT
      ST COUNT
      LD MyFIFO.PREAD
      ST PREAD
      LD MyFIFO.PWRITE
      ST PWRITE
```



2.25. File Management functions

The following functions enable sequential read / write operations in disk files:

F_ROPEN	open a file for reading
F_WOPEN	create or reset a file and open it for writing
F_AOPEN	create or open a file in append mode
F_CLOSE	close an open file
F_EOF	test if the end of file is reached in a file open for read
FA_READ	read a DINT integer from a binary file
FA_WRITE	write a DINT integer to a binary file
FM_READ	read a STRING value from a text file
FM_WRITE	write a STRING value to a text file
FB_READ	read binary data from a file
FB_WRITE	write binary data to a file
F_EXIST	test if a file exist
F_GETSIZE	get the size of a file
F_COPY	copy a file
F_DELETE	remove a file
F_RENAME	rename a file

Each file is identified in the application by a unique handle manipulated as a DINT value. The file handles are allocated by the target system. Handles are returned by the Open functions and used in all other calls for identifying the file.

Note:

- File are opened and closed directly by the Operating System of the target. Opening some files may be dangerous for system safety and integrity. The number of open files may be limited by the target system.
- Opening a file may be unsuccessful (invalid path or file name, too many open files...) Your application should process such error cases in a safe way.
- File management may be not available on some targets. Please refer to OEM instructions for further details about available features.
- Valid paths for storing files depend on the target implementation. Please refer to OEM instructions for further details about available paths.
- Using the simulator, all pathnames are ignored, and files are stored in a reserved directory. Only the file name passed to the Open functions is taken into account.



2.25.1. F_ROOPEN

Function	Open a file for reading
Command	ID := F_ROOPEN (PATH);
Inputs	PATH : STRING Name of the file - the file must exist May include a path name according to target system conventions ID : DINT ID of the open file or <i>NULL</i> if the file cant be read

2.25.2. F_WOPEN

Function	Open a file for writing
Command	ID := F_WOPEN (PATH);
Inputs	PATH : STRING Name of the file May include a path name according to target system conventions ID : DINT ID of the open file or <i>NULL</i> if the file cant be open.
Remarks	If the file does not exist, it is created. If the file already exists, its contents is cleared.

2.25.3. F_AOPEN

Function	Open a file in <i>append</i> mode
Command	ID := F_WOPEN (PATH);
Inputs	PATH : STRING Name of the file. may include a pathname according to target system conventions. ID : DINT ID of the open file or <i>NULL</i> if the file cant be open.
Remarks	If the file does not exist, it is created. If the file already exists, it is open at the end for append.

2.25.4. F_CLOSE

Function	Close an open file
Command	OK := F_CLOSE (ID);
Inputs	ID : DINT ID of the open file. OK : BOOL Return check: <i>TRUE</i> if successful.



2.25.5. F_EOF

Function	Test if end of file is encountered
Command	OK := F_EOF (ID);
Inputs	ID : DINT ID of a file open for reading OK : BOOL <i>TRUE</i> if the end of file has been encountered
Remarks	F_EOF must be used only for files open in read mode by the F_OPEN function.

2.25.6. FA_READ

Function	Read a DINT value from a file
Command	Q := FA_READ (ID);
Inputs	ID : DINT ID of a file open for reading Q : DINT Read value or 0 in case of error
Remarks	Integer values read by FA_READ must have been written by the FA_WRITE function. Integers are stored in binary format in the file, using memory conventions of the target system.

2.25.7. FA_WRITE

Function	Write a DINT value to a file
Command	OK := FA_WRITE (ID, IN);
Inputs	ID : DINT ID of a file open for writing IN : DINT integer value to be written OK : BOOL return check: <i>TRUE</i> if successful
Remarks	Integers are stored in binary format in the file, using memory conventions of the target system.

2.25.8. FM_READ

Function	Read a STRING value from a file
Command	Q := FM_READ (ID);
Inputs	ID : DINT ID of a file open for reading Q : STRING read value or empty string in case of error
Remarks	This function is intended to read a text line in the file. Reading stops when end of line character is encountered. Reading stops when the maximum length declared for the return variable is reached.



2.25.9. FM_WRITE

Function	Write a STRING value to a file
Command	OK := FM_WRITE (ID, IN);
Inputs	ID : DINT ID of a file open for writing IN : STRING string value to be written OK : BOOL return check: <i>TRUE</i> if successful
Remarks	This function writes a text line in the file. End of line character is systematically written after the input string.

2.25.10. FB_READ

Function	Read binary data from a file
Command	OK := FB_READ (ID, V);
Inputs	ID : DINT ID of a file open for reading V : ANY variable to be read (cannot be string) OK : BOOL return check: <i>TRUE</i> if successful
Remarks	Variables are stored in binary format in the file, using memory conventions of the target system.

2.25.11. FB_WRITE

Function	Write binary data to a file
Command	OK := FB_WRITE (ID, V);
Inputs	ID : DINT ID of a file open for writing V : ANY variable to be written (cannot be string) OK : BOOL return check: <i>TRUE</i> if successful
Remarks	Variables are stored in binary format in the file, using memory conventions of the target system.

2.25.12. F_EXIST

Function	Test if file exist
Command	OK := F_EXIST (PATH);
Inputs	PATH : STRING name of the file may include a pathname according to target system conventions OK : BOOL <i>TRUE</i> if the file exists
Remarks	Variables are stored in binary format in the file, using memory conventions of the target system.



2.25.13. F_GETSIZE

Function	Get file size	
Command	<code>SIZE := F_GETSIZE (PATH);</code>	
Inputs	PATH : STRING	name of the file. May include a pathname according to target system conventions
	SIZE : DINT	Size of the file in bytes

2.25.14. F_COPY

Function	Copy a file	
Command	<code>OK := F_COPY (SRC, DST);</code>	
Inputs	SRC : STRING	Name of the source file (must exist). May include a pathname according to target system conventions.
	DST : STRING	Name of the destination file. May include a pathname according to target system conventions.
	OK : BOOL	<i>TRUE</i> if successful.

2.25.15. F_DELETE

Function	Remove a file	
Command	<code>OK := F_DELETE (PATH);</code>	
Inputs	PATH : STRING	Name of the file (must exist). May include a pathname according to target system conventions
	OK : BOOL	<i>TRUE</i> if successful

2.25.16. F_RENAME

Function	Rename a file	
Command	<code>OK := F_RENAME (PATH, NEWNAME);</code>	
Inputs	PATH : STRING	Name of the file (must exist). May include a pathname according to target system conventions
	NEWNAME : STRING	New name for the file
	OK : BOOL	<i>TRUE</i> if successful



2.25.17. GETSYSINFO

Function Block Returns system Information

Inputs INFO : DINT Identifier of the requested Information

Outputs Q : DINT Value of the requested Information or **0** if error

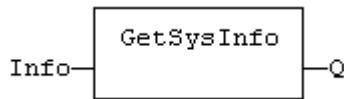
Remarks The INFO parameter can be one of the following predefined values:

Value	Description
<code>_SYSINFO_TRIGGER_MICROS</code>	Programmed cycle time in micro-seconds.
<code>_SYSINFO_TRIGGER_MS</code>	Programmed cycle time in milliseconds.
<code>_SYSINFO_CYCLETIME_MICROS</code>	Duration of the previous cycle in micro-seconds.
<code>_SYSINFO_CYCLETIME_MS</code>	Duration of the previous cycle in milliseconds.
<code>_SYSINFO_CYCLEMAX_MICROS</code>	Maximum detected cycle time in micro-seconds.
<code>_SYSINFO_CYCLEMAX_MS</code>	Maximum detected cycle time in milliseconds.
<code>_SYSINFO_CYCLESTAMP_MS</code>	Time stamp of the current cycle in milliseconds (OEM dependent).
<code>_SYSINFO_CYCLEOVERFLOWS</code>	Number of detected cycle time overflows.
<code>_SYSINFO_CYCLECOUNT</code>	Counter of cycles.
<code>_SYSINFO_APPVERSION</code>	Version number of the application.
<code>_SYSINFO_APPSTAMP</code>	Compiling date stamp of the application.
<code>_SYSINFO_CODECRC</code>	CRC of the application code.
<code>_SYSINFO_DATACRC</code>	CRC of the application symbols.
<code>_SYSINFO_FREEHEAP</code>	Available space in memory heap (bytes)
<code>_SYSINFO_DBSIZE</code>	Space used in RAM (bytes)
<code>_SYSINFO_ELAPSED</code>	Seconds elapsed since startup

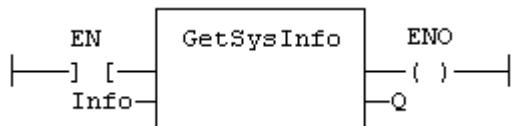
In LD language, the operation is executed only if the input rung (EN) is *TRUE*. The output rung (ENO) keeps the same value as the input rung. In IL, the input must be loaded in the current result before calling the function.

ST Language `Q := GETSYSINFO (INFO);`

FBD Language



LD Language The function is executed only if EN is *TRUE*. ENO keeps the same value as EN



IL Language `Op1: LD INFO
GETSYSINFO
ST Q`

2.25.18. HYSTER

Function Block Hysteresis detection

Inputs

XIN1 : REAL	First input signal
XIN2 : REAL	Second input signal
EPS : REAL	Hysteresis

Outputs

Q : BOOL	Detected hysteresis: <i>TRUE</i> if XIN1 becomes greater than XIN2+EPS and is not yet below XIN2-EPS
----------	--

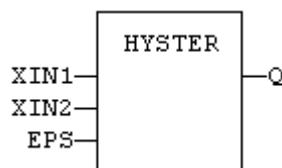
Remarks The hysteresis is detected on the difference of XIN1 and XIN2 signals. In LD language, the input rung (EN) is used for enabling the block. The output rung is the Q output.

ST Language MyHyst is a declared instance of HYSTER function block

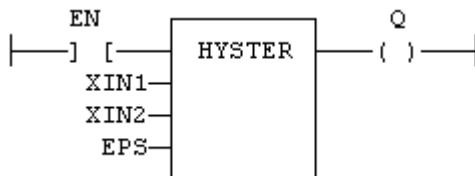
MyHyst (XIN1, XIN2, EPS);

Q := MyHyst.Q;

FBD Language



LD Language The block is not called if EN is *FALSE*



IL Language MyHyst is a declared instance of HYSTER function block.

Op1: CAL MyHyst (XIN1, XIN2, EPS)

LD MyHyst.Q

ST Q



2.25.19. INTEGRAL

Function Block Calculates the integral of a signal

Inputs RUN : BOOL Run command: *TRUE*=integrate / *FALSE*=hold

R1 : BOOL Overriding reset

XIN : REAL Input signal

X0 : REAL Initial value

CYCLE : TIME Sampling period (should not be less than the target cycle timing)

Outputs Q : DINT Running mode report: NOT (R1)

XOUT : REAL Output signal

Remarks In LD language, the input rung is the RUN command. The output rung is the Q report status.

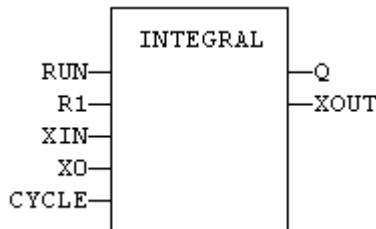
ST Language MyIntg is a declared instance of INTEGRAL function block

```
MyIntg (RUN, R1, XIN, X0, CYCLE );
```

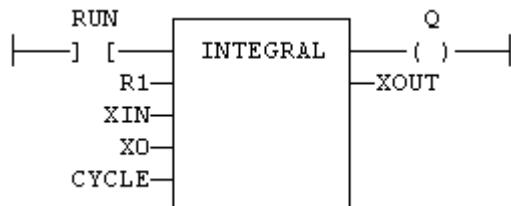
```
Q := MyIntg.Q;
```

```
XOUT := MyIntg.XOUT;
```

FBD Language



LD Language



IL Language MyIntg is a declared instance of INTEGRAL function block.

```
Op1: CAL MyIntg (RUN, R1, XIN, X0, CYCLE )
```

```
LD MyIntg.Q
```

```
ST Q
```

```
LD MyIntg.XOUT
```

```
ST XOUT
```



2.25.20. LIFO

Function Block Manages a last in / first out stack

Inputs	PUSH : BOOL	Push a new value (on rising edge)
	POP : BOOL	Pop a new value (on rising edge)
	RST : BOOL	Reset the list
	NEXTIN : ANY	Value to be pushed
	NEXTOUT : ANY	Value at the top of the stack - updated after call!
	BUFFER : ANY	Array for storing values

Outputs	EMPTY : BOOL	<i>TRUE</i> if the stack is empty
	OFLO : BOOL	<i>TRUE</i> if overflow on a PUSH command
	COUNT : DINT	Number of values in the stack
	PREAD : DINT	Index in the buffer of the top of the stack
	PWRITE : DINT	Index in the buffer of the next push position

Remarks NEXTIN, NEXTOUT and BUFFER must have the same data type and cannot be STRING.

The NEXTOUT argument specifies a variable that is filled with the value at the top of the stack after the block is called.

Values are stored in the BUFFER array. Data is never shifted or reset. Only read and write pointers and pushed values are updated. The maximum size of the stack is the dimension of the array.

The first time the block is called, it remembers on which array it should work. If you call later the same instance with another BUFFER input, the call is considered as invalid and makes nothing. Outputs reports an empty stack in this case.

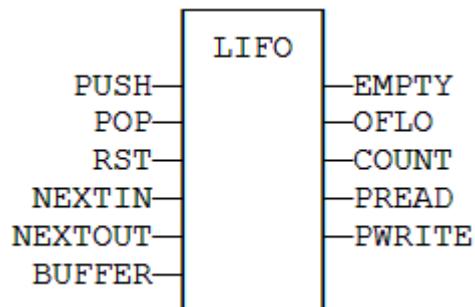
In LD language, input rung is the PUSH input. The output rung is the EMPTY output.

ST Language MyLIFO is a declared instance of LIFO function block

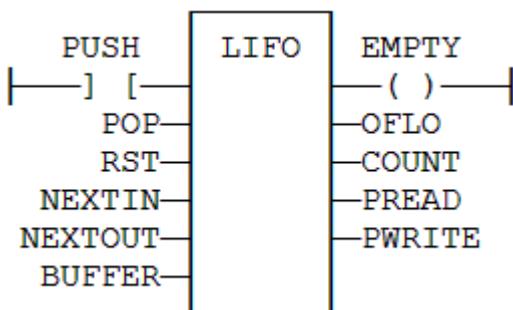
```
MyLIFO (PUSH, POP, RST, NEXTIN, NEXTOUT, BUFFER );
EMPTY := MyLIFO.EMPTY;
OFLO := MyLIFO.OFLO;
COUNT := MyLIFO.COUNT;
PREAD := MyLIFO.PREAD;
PWRITE := MyLIFO.PWRITE;
```



FBD Language



LD Language



IL Language

MyLIFO is a declared instance of LIFO function block.

Op1: CAL MyLIFO (PUSH, POP, RST, NEXTIN, NEXTOUT, BUFFER)

```
LD MyLIFO.EMPTY
ST EMPTY
LD MyLIFO.OFLO
ST OFLO
LD MyLIFO.COUNT
ST COUNT
LD MyLIFO.PREAD
ST PREAD
LD MyLIFO.PWRITE
ST PWRITE
```

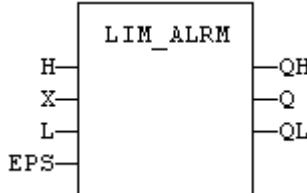
2.25.21. LIM_ALRM

Function Block Detects High and Low limits of a signal with hysteresis

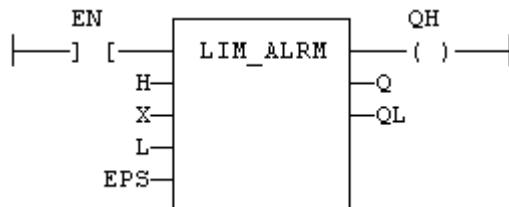
Inputs	H : REAL	Value of the High limit
	X : REAL	Input signal
	L : REAL	Value of the Low limit
	EPS : REAL	Value of the hysteresis
Outputs	QH : BOOL	<i>TRUE</i> if the signal exceeds the High limit
	Q : BOOL	<i>TRUE</i> if the signal exceeds one of the limits (equals to QH OR QL)
	QL : BOOL	<i>TRUE</i> if the signal exceeds the Low limit
Remarks	In LD language, the input rung (EN) is used for enabling the block. The output rung is the QH output.	
ST Language	MyAlarm is a declared instance of LIM_ALRM function block.	

```
MyAlarm (H, X, L, EPS );  
QH := MyAlarm.QH;  
Q := MyAlarm.Q;  
QL := MyAlarm.QL;
```

FBD Language



LD Language The block is not called if EN is *FALSE*.



IL Language MyAlarm is a declared instance of LIM_ALRM function block.

Op1: CAL MyAlarm (H, X, L, EPS)

```
LD MyAlarm.QH  
ST QH  
LD MyAlarm.Q  
ST Q  
LD MyAlarm.QL  
ST QL
```



2.25.22. LogFileCSV

Function Block Generate a log file in CSV format for a list of variables

Inputs	LOG : BOOL	Variables are saved on any rising edge of this input
	RST : BOOL	Reset the contents of the CSV file
	LIST : DINT	ID of the list of variables to log (use VLID function)
	PATH : STRING	Pathname of the CSV file

Outputs	Q : BOOL	<i>TRUE</i> if the requested operation has been performed without error
	ERR : DINT	Error report for the last requested operation (0 is OK)

Note:

- File are opened and closed directly by the Operating System of the target. Opening some files may be dangerous for system safety and integrity. The number of open files may be limited by the target system.
- Opening a file may be unsuccessful (invalid path or file name, too many open files...) Your application should process such error cases in a safe way.
- File management may be not available on some targets.
- Valid paths for storing files depend on the target implementation.

Remarks This function enables to log values of a list of variables in a CSV file. On each rising edge of the LOG input, one more line of values is added to the file. There is one column for each variable, as they are defined in the list.

The list of variables is prepared using the workbench or a text editor. Use the VLID function to get the identifier of the list.

On a rising edge of the RST command, the file is emptied.

The "SetCsvOpt" function described at the end of this page enables you to customize the format of the CSV file.

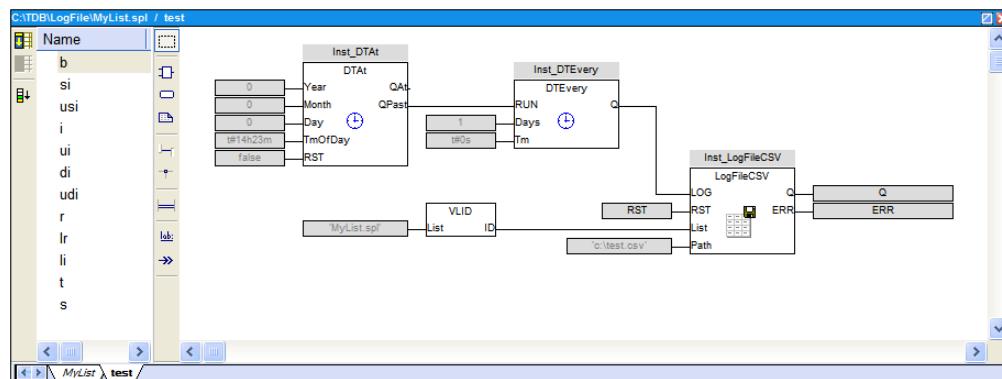
When a LOG or RST command is requested, the Q output is set to *TRUE* if successful. In case of error, a report is given in the ERR output. Possible error values are:

Value	Meaning
--------------	----------------

- | | |
|----------|---|
| 1 | Cannot reset file on a RST command. |
| 2 | Cannot open file for data storing on a LOG command. |
| 3 | Embedded lists are not supported by the runtime. |
| 4 | Invalid list ID. |
| 5 | Error while writing to file. |

Combined with real time clock management functions, this block provides a very easy way to generate a periodical log file. The following Example shows a list and a program that log values everyday at 14h23:

Example:



ST Language

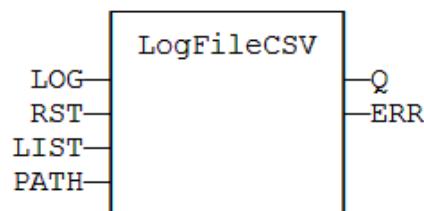
MyLOG is a declared instance ofLogFileCSV function block.

MyLOG (LOG, RST, LIST, PATH);

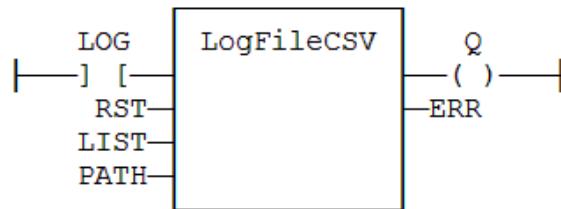
Q := MyLOG.Q;

ERR := MyLOG.ERR;

FBD Language



LD Language





2.25.23. SetCsvOpt

Function As default, the LogFileCsv block uses semicolons (";") as separators and dots (".") as decimal points. The SetCsvOpt function enables you to change these characters.

Syntax `OK := SetCsvOpt (strSeparator, strDecimal);`

Parameters StrSeparator (STRING) : single character string to be used as separator

strDecimal (STRING) : single character string to be used as decimal point

OK (BOOL) : TRUE if successful. FALSE if inputs are not single character strings.

MBMasterRTU

Function Block MODBUS-RTU Master. This function block manages a master on MODBUS-RTU over a serial line and enables the management of various MODBUS functions. Function blocks MBMASTERTCP and MBMASTERRTU give an alternative to the MODBUS configurator. You do not need to configurate anything in the fieldbus configurator to use these blocks.

Inputs CNX : BOOL A rising edge on this input to open the specified COM port. If the port is successfully open, the *Ready* output will be set to TRUE. If the opening fails, an error is reported by the *Error* output.

BREFRESH : BOOL A rising edge on this input starts a new exchange such as specified by other parameters. A exchange can be started only if the *Ready* output is TRUE.

SLAVE : DINT Slave number (or unit number) of the MODBUS TCP server.

FCODE : DINT MODBUS function code. The following functions are supported:

- 1 = Read coil bits
- 2 = Read input bits
- 3 = Read holding registers
- 4 = Read input registers
- 5 = Write one coil bit
- 6 = Write one holding registers
- 7 = Read exception status
- 15 = Write N coil bits
- 16 = Write N holding registers
- 17 = Report slave ID
- 23 = Read/write holding registers

ADDR : DINT Address of the first address to read. Used only for functions 1, 2, 3, 4 and 23.

NBR : DINT Number of items to read. Used only for functions 1, 2, 3, 4 and 23.



ADDW : DINT	Address of the first address to write. Used only for functions 5, 6, 15, 16 and 23.
NBW : DINT	Number of items to write. Used only for functions 5, 6, 15, 16 and 23. Must be 1 for functions 5 and 6.
DATA : ARRAY OF ...	Buffer containing the values read or written. Must be set on input before calling functions 5, 6, 15, 16 and 23. Filled on for other functions and also for function 23. The dimension of the array must be greater than or equal to the specified number of items. The data type of the array depends on the function: BOOL for functions 1, 2, 5 and 15 INT or UINT or WORD for functions 3, 4, 6, 16 and 23 SINT or USINT or BYTE for functions 7 and 17
TIMEOUT : TIME	Communication timeout for the complete exchange.
PORT : STRING	Specification of the COM port (e.g. 'COM1:9600,N,8,1').
Outputs	READY : BOOL This output is TRUE when the connection is correctly established and no pending exchange is currently running. OK : BOOL This output is set to TRUE at the end of a successful exchange. It is reset to FALSE when a new exchange is started. ERROR : DINT Error report. Can be one of the following values: 1 = invalid function code 2 = bad address 3 = bad value 4 = server failed 5 = server acknowledge 6 = server is busy 8 = data parity error 10 = gateway bad path 11 = gateway target failed 128 = timeout on exchange 129 = bad CRC 130 = frame error or bad COM port 200 = invalid slave number 201 = invalid MODBUS address 202 = invalid number of items or DATA array too small 203 = invalid type for DATA array 204 = invalid number of items

Note:

some devices may answer some other specific error reports



2.25.24. MBMasterTCP

Function Block MODBUS-TCP Master (client). This function block manages a client connection on MODBUS-TCP and enables the management of various MODBUS functions. Function blocks MBMASTERTCP and MBMASTERRTU give an alternative to the MODBUS configurator. You do not need to configurate anything in the fieldbus configurator to use these blocks.

Inputs	CNX : BOOL	A rising edge on this input establishes the connection with the specified server. Later on, if the connection is successful, the <i>Ready</i> output will be set to TRUE. If the connection fails, an error is reported by the <i>Error</i> output. After the connection is lost (typically when unplugging the Ethernet cable) the application is responsible for rising again the <i>Cnx</i> input in order to attempt a new connection.
	BREFRESH : BOOL	A rising edge on this input starts a new exchange such as specified by other parameters. A exchange can be started only if the <i>Ready</i> output is TRUE.
	SLAVE : DINT	Slave number (or unit number) of the MODBUS TCP server.
	FCODE : DINT	MODBUS function code. The following functions are supported: 1 = Read coil bits 2 = Read input bits 3 = Read holding registers 4 = Read input registers 5 = Write one coil bit 6 = Write one holding registers 7 = Read exception status 15 = Write N coil bits 16 = Write N holding registers 17 = Report slave ID 23 = Read/write holding registers
	ADDR : DINT	Address of the first address to read. Used only for functions 1, 2, 3, 4 and 23.
	NBR : DINT	Number of items to read. Used only for functions 1, 2, 3, 4 and 23.
	ADDW : DINT	Address of the first address to write. Used only for functions 5, 6, 15, 16 and 23.
	NBW : DINT	Number of items to write. Used only for functions 5, 6, 15, 16 and 23. Must be 1 for functions 5 and 6.
	DATA : ARRAY OF ...	Buffer containing the values read or written. Must be set on input before calling functions 5, 6, 15, 16 and 23. Filled on for other functions and also for function 23. The dimension



of the array must be greater than or equal to the specified number of items. The data type of the array depends on the function:

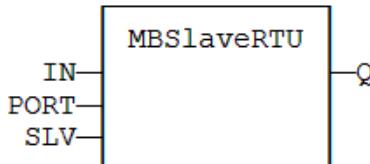
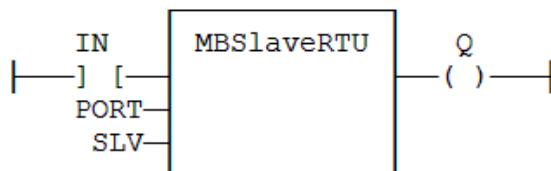
BOOL for functions 1, 2, 5 and 15
INT or UINT or WORD for functions 3, 4, 6, 16 and 23
SINT or USINT or BYTE for functions 7 and 17

	TIMEOUT : TIME	Communication timeout for the complete exchange.
	IP : STRING	IP address of the MODBUS server.
	PORT : DINT	TCP port number of the server (usually 502).
Outputs	READY : BOOL	This output is TRUE when the connection is correctly established and no pending exchange is currently running.
	OK : BOOL	This output is set to TRUE at the end of a successful exchange. It is reset to FALSE when a new exchange is started.
	ERROR : DINT	Error report. Can be one of the following values: 1 = invalid function code 2 = bad address 3 = bad value 4 = server failed 5 = server acknowledge 6 = server is busy 8 = data parity error 10 = gateway bad path 11 = gateway target failed 128 = timeout on exchange 129 = bad CRC 130 = frame error or bad COM port 200 = invalid slave number 201 = invalid MODBUS address 202 = invalid number of items or DATA array too small 203 = invalid type for DATA array 204 = invalid numner of items

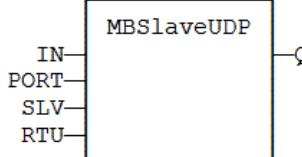
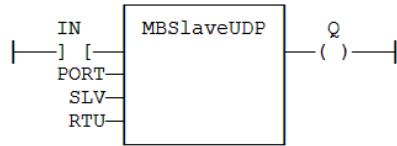
Note:

Some devices may answer some other specific error reports

2.25.25. MBSlaveRTU / MBSlaveRTUex

Function	MODBUS RTU Slave protocol on serial port
Inputs	IN : BOOL Enabling command: the port is open when this input is <i>TRUE</i> PORT : STRING Settings string for the serial port (e.g. 'COM1:9600,N,8,1') SLV : DINT MODBUS slave number SrvID : STRING MBSlaveRTUex only: ID of the server in the configuration
Outputs	Q : BOOL <i>TRUE</i> if the port is successfully open
Remarks	When active, this function block manages the MODBUS RTU Slave protocol on the specified serial communication port. The configuration of the MODBUS Slave map (designing MODBUS addresses) is done using the MODBUS configuration tool, from the Fieldbus Configurator. There can be several instances of the MBSlaveRTU working simultaneously on different serial ports. Other MODBUS Slave connections (TCP server, UDP) can also be active at the same time. The slave number entered in the MODBUS Slave configuration tool is ignored when MODBUS Slave protocol is handled by this function block. Instead, the SLV input specifies the MODBUS slave number. With MBSlaveRTUex, you can specify the ID of a server from the MODBUS slave configuration. If you specify an empty string, the first server is used. MBSlaveRTU always works with the first server in the configuration.
ST Language	MySlave is a declared instance of MBSlaveRTU function block. MySlave (IN, PORT, SLV); Q := MySlave.Q;
FBD Language	
LD Language	
IL Language	MySlave is a declared instance of MBSlaveRTU function block. Op1: CAL MySlave (IN, PORT, SLV) LD MySlave.Q ST Q LD MyCounter.CV ST CV

2.25.26. MBSlaveUDP / MBSlaveUDPEx

Function	MODBUS UDP Slave protocol on ETHERNET										
Inputs	<table><tr><td>IN : BOOL</td><td>Enabling command: the port is open when this input is <i>TRUE</i></td></tr><tr><td>PORT : DINT</td><td>ETHERNET port number</td></tr><tr><td>SLV : DINT</td><td>MODBUS slave number</td></tr><tr><td>RTU : BOOL</td><td>Protocol: <i>TRUE</i> = MODBUS RTU / <i>FALSE</i> = Open MODBUS</td></tr><tr><td>SrvID : STRING</td><td>MBSlaveUDPEx only: ID of the server in the configuration</td></tr></table>	IN : BOOL	Enabling command: the port is open when this input is <i>TRUE</i>	PORT : DINT	ETHERNET port number	SLV : DINT	MODBUS slave number	RTU : BOOL	Protocol: <i>TRUE</i> = MODBUS RTU / <i>FALSE</i> = Open MODBUS	SrvID : STRING	MBSlaveUDPEx only: ID of the server in the configuration
IN : BOOL	Enabling command: the port is open when this input is <i>TRUE</i>										
PORT : DINT	ETHERNET port number										
SLV : DINT	MODBUS slave number										
RTU : BOOL	Protocol: <i>TRUE</i> = MODBUS RTU / <i>FALSE</i> = Open MODBUS										
SrvID : STRING	MBSlaveUDPEx only: ID of the server in the configuration										
Outputs	Q : BOOL <i>TRUE</i> if the port is successfully open										
Remarks	<p>When active, this function block manages the MODBUS UDP Slave protocol on the specified ETHERNET port. The configuration of the MODBUS Slave map (designing MODBUS addresses) is done using the MODBUS configuration tool, from the Fieldbus Configurator.</p> <p>There can be several instances of the MBSlaveUDP working simultaneously on different serial ports. Other MODBUS Slave connections (TCP server, serial) can also be active at the same time.</p> <p>The slave number entered in the MODBUS Slave configuration tool is ignored when MODBUS Slave protocol is handled by this function block. Instead, the SLV input specifies the MODBUS slave number. If SLV is 0 then the default port number from the MODBUS configuration is used.</p> <p>With MBSlaveUDPEx, you can specify the ID of a server from the MODBUS slave configuration. If you specify an empty string, the first server is used. MBSlaveUDP always works with the first server in the configuration.</p>										
ST Language	MySlave is a declared instance of MBSlaveUDP function block. <code>MySlave (IN, PORT, SLV, RTU); Q := MySlave.Q;</code>										
FBD Language											
LD Language											
IL Language	MySlave is a declared instance of MBSlaveUDP function block. <code>Op1: CAL MySlave (IN, PORT, SLV, RTU) LD MySlave.Q ST Q LD MyCounter.CV ST CV</code>										



Handling

MAN-IEC61131-3PUG (Ver. 1.001)

2.26. MQTT (MQ Telemetry Transport) Protocol Handling

The system includes a set of functions and blocks for managing the MQTT protocol from IEC applications. Functions and blocks described here enable the establishment of MQTT connections, subscribing and publishing MQTT message. Additional functions are provided for passing IEC variable values in message data.

MQtt functions are designed for working with text buffers. Text buffers are used for storing MQTT message data.



Attention

MQTT message topic names are stored in STRING variables and thus cannot exceed 255 characters.

There must be one instance of the TxbManager declared in your application for using text buffers related functions.

Below are the functions and function blocks for handing MQTT connections and messages:

Connection and status

MQttConnection Handles a connection to a MQTT message broker.

MQttLastError Get detailed error report about last call.

MQttMsgStatus Get current status of a message.

Messaging

MQttPublish Published a message.

MQttSubscribe Subscribe to a message.

MQttUnsubscribe Unsubscribe to a message.

MQttReceivePub Get received messages.

MQttNbRec Get number of messages received.

Additional functions

MQttIECFormat Format an IEC value in message data.

MQttIECParse Parse an IEC value in message data.

The following sections describe the functions of the MQTT protocol.



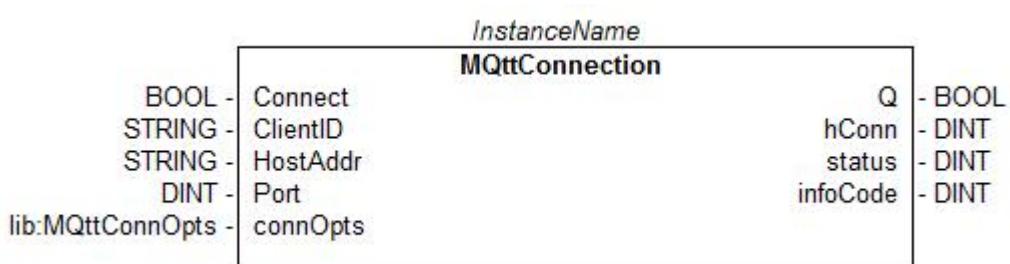
2.27. MQTT Connection and Status

2.27.1. MQttConnection

Function block This function block is used for establishing a connection to a message broker (server). When connection is established, the "hConn" output is the handle of the connection, to be passed to other functions in charge of messaging.

Inputs	Connect : BOOL	A connection is established when this input changes from <i>FALSE</i> to <i>TRUE</i> . See status output to know the connection status.
		The connection is released when this input changes from <i>TRUE</i> to <i>FALSE</i> .
	clientID : STRING	ID of the client application. If you establish several connections at the same time, each connection must be identified by a unique client ID.
	HostAddr : STRING	Address of the broker.
	Port : DINT	IP port of the broker.
	connOpts : lib:MQttConnOpts	Connection options (see details below). The MQttConnOpts type is a structure defined in library
Outputs	Q : BOOL	<i>TRUE</i> when connection is established. This output simply indicates that the connection procedure is finished. Connection is not ensured to be OK. See the "status" output to know the state of the current connection.
	hConn : DINT	Handle of the connection, that must be passed to other MQtt functions to identify the MQTT connection.
	status : DINT	Current status of the connection. See details below.
	InfoCode : DINT	Additional Information about current connection status.

Diagram





ST Language The MQttConnOpts type is a structure defined in library and has the following form:

```
STRUCT
    retryInterval : DINT;
    keepAlive : DINT;
    willData : DINT;
    willTopic : STRING;
    cleanStart : BOOL;
    bWill : BOOL;
END_STRUCT
```

Variables	retryInterval	Interval in seconds at which messages should be retried if a send fails.
	KeepAlive	A length of time in seconds. If the MQTT server does not receive data within this time limit it will assume the client application has terminated.
	willData	Handle of a text buffer containing the will message data, if specified.
	willTopic	Will topic name if specified
	cleanStart	If <i>TRUE</i> , it means that when the application disconnects cleanly or otherwise, the broker will clean up on behalf of the application, removing all active subscriptions and any outstanding data for that connection.
	bWill	Indicates if a will message is specified. The will message will be published in the event of the unexpected termination of this application

Connection Status Below are the possible values for the status of the connection. You must check the status after asking for a connection to be established:

0	Connection status unknown - please wait
2009	Trying to find MQTT server - please wait
6	Connecting to the server - please wait
7	Connection is OK
8	Disconnected
9	Connection was broken

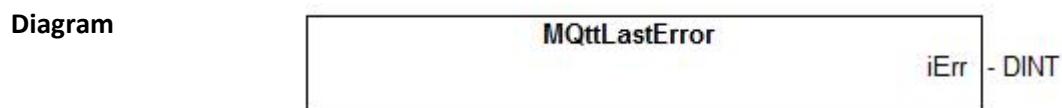
The status can also be an error code if connection fails or in case of invalid parameters. See description of the MQttLastError function for further information.



2.27.2. MQttLastError

Function block Most of MQTT functions and blocks simply return a Boolean Information as a return value. This function can be called after any other function giving a *FALSE* return. It gives a detailed error code about the last detected error.

Outputs iErr : DINT Error code reported by the last call:
 $0 = \text{OK}$
 $\text{other} = \text{error}$



Connection Status Below are possible error codes:

- | | |
|------|---|
| 1001 | MQTT protocol version mismatch |
| 1002 | Hostname not found |
| 1003 | MQTT queues overflow |
| 1004 | MQTT messaging failed |
| 1005 | Timeout |
| 1006 | MQTT message overflow |
| 1007 | Message persistence failed |
| 1008 | Invalid connection handle |
| 1010 | Will topic not supported |
| 1011 | Internal error |
| 1012 | Frame error |
| 1013 | Data is too big |
| 1014 | MQTT is already connected |
| 1019 | Bad client ID |
| 1020 | Broker is unavailable |
| 1021 | MQTT socket closed |
| 1022 | Out of memory |
| 2001 | Cannot start MQ tasks Cannot start MQTT tasks |
| 2002 | Connection already established |



2003	No connection established
2004	Invalid text buffer handle
2005	Text buffers are not initialized
2006	Library not licensed
2007	Too many open connections
2008	Invalid connection for reading pubs
2009	Connection in progress - not ready yet
2010	Client ID used more than once
2011	Internal error
1001	MQTT protocol version mismatch
1002	Hostname not found
1003	MQTT queues overflow

2.27.3. MQttMsgStatus

Function This function returns the current status of a MQTT message. The message is identified by a handle returned by other MQTT functions (publish, subscribe...)

Inputs hConn : DINT Handle of the MQTT connection, returned by MQttConnection block

hMsg : DINT Handle of the message

Outputs status : DINT Current status of the message.

Diagram



Connection Status Possible status values are:

- | | |
|---|------------------------|
| 1 | Message delivered |
| 2 | Retrying |
| 3 | In progress |
| 4 | Invalid message handle |



2.28. MQTT Messaging

2.28.1. MQttPublish

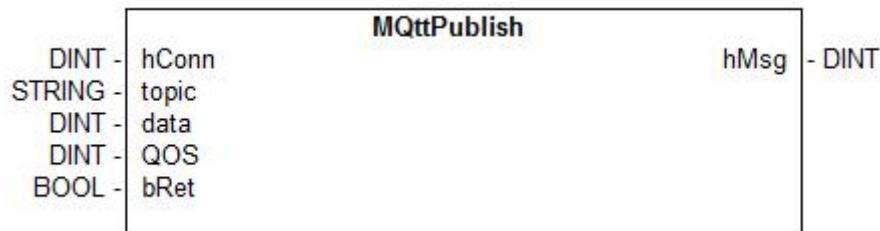
Function This function publishes a MQTT message on a valid connection.



The **data** text buffer is no more used after the call to the function. The application is responsible for releasing the text buffer after the call if no more needed, in order to avoid memory leaks.

Inputs	hConn : DINT	Handle of the MQTT connection, returned by MQttConnection block
	topic : STRING	Message topic name
	data : DINT	Handle of a text buffer containing message data
	QOS : DINT	Quality of service (0 or 1 or 2)
	bRet : BOOL	If <i>TRUE</i> , the message will be retained by the MQTT borker until another publication is received for the same topic
Outputs	hMsg : DINT	Handle of the message. This handle can be passed to the MQttMsgStatus function to track the status of the message.

Diagram



2.28.2. MQttSubscribe

Function This function subscribes to a MQTT message.

Inputs

hConn : DINT	Handle of the MQTT connection, returned by MQttConnection block.
topic : STRING	Message topic name.
QOS : DINT	Quality of service (0 or 1 or 2).

Outputs

hMsg : DINT	Handle of the message. This handle can be passed to the MQttMsgStatus function to track the status of the message.
-------------	--

Diagram



2.28.3. MQttUnsubscribe

Function This function unsubscribes to a MQTT message.

Inputs

hConn : DINT	Handle of the MQTT connection, returned by MQttConnection block.
topic : STRING	Message topic name.

Outputs

hMsg : DINT	Handle of the message. This handle can be passed to the MQttMsgStatus function to track the status of the message.
-------------	--

Diagram



2.28.4. MQttReceivePub

Function

Use this function block to pump messages received on a valid MQTT connection.



The *data* text buffer is allocated by the function block if a new message is received. The application is responsible for releasing the text buffer after the call when no more needed, in order to avoid memory leaks.

Received messages with higher quality of services are returned first.

The message is "poped" from a FIFO internally managed by MQTT function blocks, and thus must be managed immediately after the call.

Inputs

hConn : DINT Handle of the MQTT connection, returned by MQttConnection block.

Outputs

Q : BOOL *TRUE* if a message is available. In that case, you must handle the message immediately after the call and release the text buffer associated to message data.

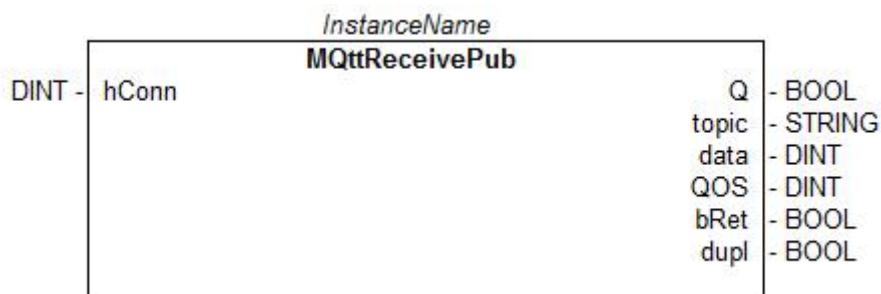
topic : STRING Message topic name.

data : DINT Handle of a new text buffer filled with message data. The application is responsible for releasing text buffer memory.

QOS : DINT Quality of service originally given to the message.

bRet : BOOL Indicates a message retained by the server.

dupl : BOOL Indicates a duplicated message.

Diagram

2.28.5. MQttNbRec

Function

This function returns the number of messages received not pumped yet by the application. The number returned matches the quality of services specified in the QOS input argument. Call the MQttReceivePub function block for pumping received messages.

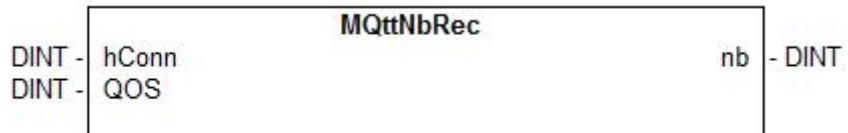
Inputs

hConn : DINT Handle of the MQTT connection, returned by MQttConnection block.

QOS : DINT Desired quality of service (0, 1, 2) or -1 for all messages.

Outputs

nb : DINT Number of messages received, according to the specified quality of service:
QOS=0 : received messages with quality 0
QOS=1 : received messages with quality 1
QOS=2 : received messages with quality 2
QOS=-1 : all received messages

Diagram



2.29. Additional MQTT functions

2.29.1. MQttIECFormat

Function This function allocates a new text buffer and fills it with the formatted IEC value of the variable connected to the IN input. See formatting specifications below.



The function returns a new allocated text buffer. The application is responsible for releasing the text buffer memory when no more needed.

Inputs IN : ANY Any single variable. This input must be directly linked to the variable.

Outputs data : DINT New allocated text buffer containing the formatted message data.

Message Format The value is formatted on a single line text, using the following format:

type:value

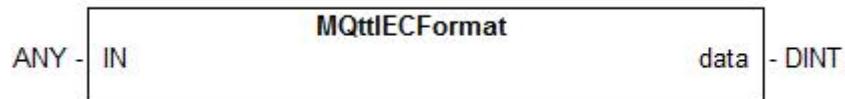
The type specified conforms to IEC convention (BOOL, DINT, REAL...). Exceptions are BYTE, WORD, DWORD that are marked as respectively USINT, UINT and UDINT.

The value is expressed in decimal form for numbers, and with the following format for special data types:

STRING: No quote is appended.

TIME: Expressed as a number of milliseconds.

Diagram





2.29.2. MQttIECParse

Function

This function parses the contents of a message data containing an IEC value, and assigns the value to the variable connected to the @OUT input.

Attention

The text buffer memory is not released by this function.

See description of the MQttIECFormat function for exact specification of message format.

Inputs

data : DINT Handle of the XML tag.

@OUT : ANY Must be directly linked to the variable that will be forced to the value specified in the message.

Outputs

Q : BOOL TRUE if the text buffer contains a valid IEC value.

Message Format

The value is formatted on a single line text, using the following format:

type:value

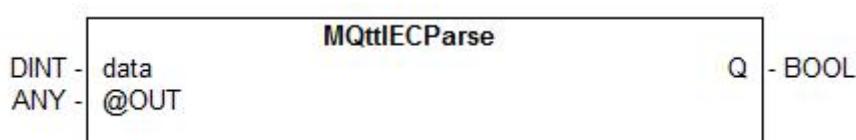
The type specified conforms to IEC convention (BOOL, DINT, REAL...). Exceptions are BYTE, WORD, DWORD that are marked as respectively USINT, UINT and UDINT.

The value is expressed in decimal form for numbers, and with the following format for special data types:

STRING: No quote is appended.

TIME: Expressed as a number of milliseconds.

Diagram

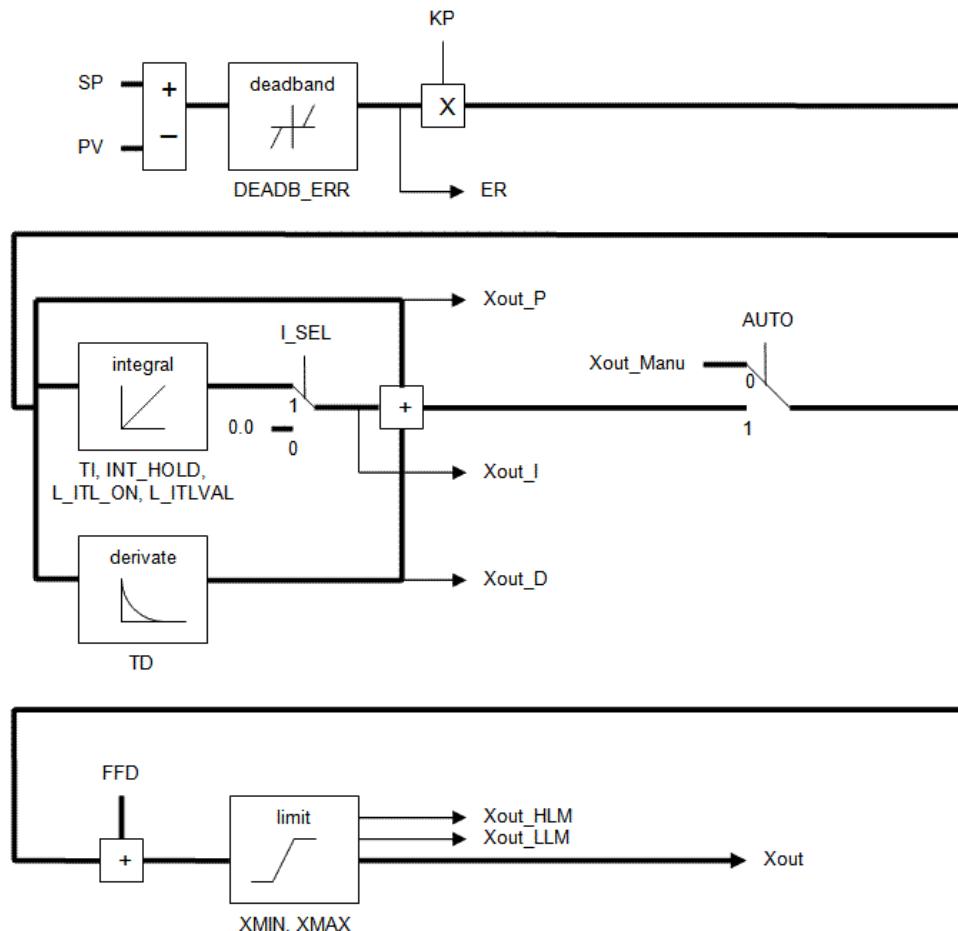




2.30. MQTT Protocol Handling (Cont.)

2.30.1. PID

Function Block	PID loop		
Inputs	AUTO	BOOL	<i>TRUE</i> = normal mode - <i>FALSE</i> = manual mode.
	PV	REAL	Process value.
	SP	REAL	Set point.
	Xout_Manu	REAL	Output value in manual mode.
	KP	REAL	Gain.
	TI	REAL	Integration time.
	TD	REAL	Derivation time.
	TS	TIME	Sampling period.
	XMIN	REAL	Minimum allowed output value.
	XMAX	REAL	Maximum output value.
	I_SEL	BOOL	If <i>FALSE</i> , the integrated value is ignored.
	INT_HOLD	BOOL	If <i>TRUE</i> , the integrated value is frozen.
	I_ITL_ON	BOOL	If <i>TRUE</i> , the integrated value is reset to I_ITLVAL.
	I_ITLVAL	REAL	Reset value for integration when I_ITL_ON is <i>TRUE</i> .
	DEADB_ERR	REAL	Hysteresis on PV. PV will be considered as unchanged if greater than (PVprev - DEADBAND_W) and less than (PRprev + DEADBAND_W).
	FFD	REAL	Disturbance value on output.
Outputs	Xout	REAL	Output command value.
	ER	REAL	Last calculated error.
	Xout_P	REAL	Last calculated proportional value.
	Xout_I	REAL	Last calculated integrated value.
	Xout_D	REAL	Last calculated derivated value.
	Xout_HLM	BOOL	<i>TRUE</i> if the output value is saturated to XMIN.
	Xout_LLM	BOOL	<i>TRUE</i> if the output value is saturated to XMAX.

Diagram**Remarks**

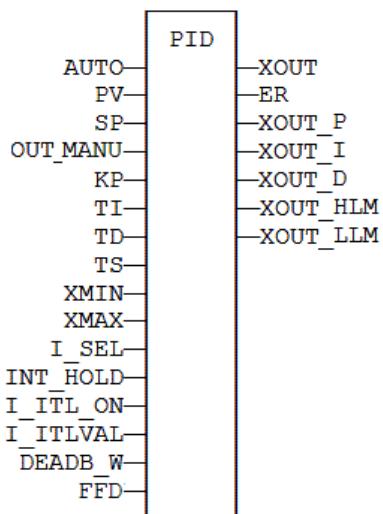
It is important for the stability of the control that the TS sampling period is much bigger than the cycle time.

In LD language, the output rung has the same value as the AUTO input, corresponding to the input rung.

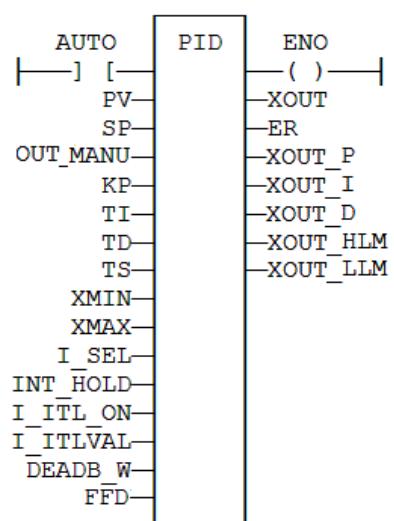
ST Language

MyPID is a declared instance of PID function block.

```
MyPID(AUTO, PV, SP, XOUT_MANU, KP, TI, TD, TS, XMIN, XMAX,  
      I_SEL, I_ITL_ON, I_ITLVAL, DEADB_ERR, FFD);  
XOUT := MyPID.XOUT;  
ER := MyPID.ER;  
XOUT_P := MyPID.XOUT_P;  
XOUT_I := MyPID.XOUT_I;  
XOUT_D := MyPID.XOUT_D;  
XOUT_HLM := MyPID.XOUT_HLM;  
XOUT_LLM := MyPID.XOUT_LLM;
```

FBD Language**LD Language**

ENO has the same state as the input rung.

**IL Language**

MyPID is a declared instance of PID function block.

**Op1: CAL MyPID (AUTO, PV, SP, XOUT_MANU, KP, TI, TD, TS,
XMIN, XMAX, I_SEL, I_ITL_ON, I_ITLVAL,
DEADB_ERR, FFD)**

```
LD MyPID.XOUT
ST XOUT
LD MyPID.ER
ST ER
LD MyPID.XOUT_P
ST XOUT_P
LD MyPID.XOUT_I
ST XOUT_I
LD MyPID.XOUT_D
ST XOUT_D
LD MyPID.XOUT_HLM
ST XOUT_HLM
LD MyPID.XOUT_LLM
```



ST XOUT_LLM



2.30.2. Printf

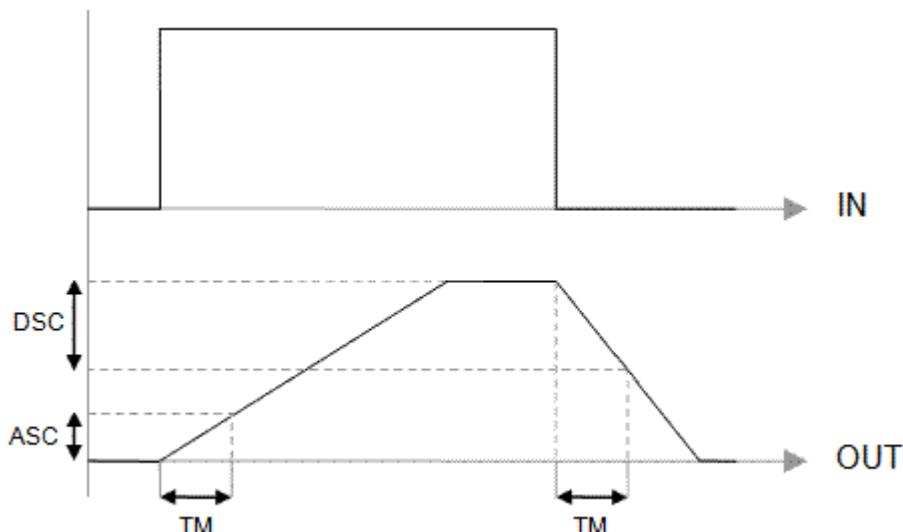
Function Block	Display a trace output	
Inputs	FMT : STRING	Trace message
	ARG1..ARG4 : DINT	Numerical arguments to be included in the trace
Outputs	Q : BOOL	Return check
Remarks	<p>This function works as the famous "printf" function of the "C" language, with up to 4 integer arguments. You can use the following pragmas in the FMT trace message to represent the arguments according to their left to the right order:</p> <p>%ld signed value in decimal %lu unsigned value in decimal %lx value in hexadecimal</p>	
	<p>The trace message is displayed in the LOG window with runtime messages. Trace is supported by the simulator.</p>	
 Attention		
	<p>Your target platform may not support trace functions. Please refer to OEM instructions for further details on available features.</p>	
Example	<pre>(* i1, i2, i3, i4 are declared as DINT *) i1 := 1; i2 := 2; i3 := 3; i4 := 4; printf ('i1=%ld; i2=%ld; i3=%ld; i4=%ld', i1, i2, i3, i4);</pre>	
	<p>Output message: i1=1; i2=2; i3=3; i4=4;</p>	

2.30.3. RAMP

Function Block Limit the ascendance or descendence of a signal

Inputs	IN : REAL	Input signal
	ASC : REAL	Maximum ascendance during time base
	DSC : REAL	Maximum descendence during time base
	TM : TIME	Time base
	RST : BOOL	Reset
Outputs	OUT : REAL	Ramp signal

Time Diagram



Remarks

Parameters are not updated constantly. They are taken into account when only:

- The first time the block is called.
- When the reset input (RST) is *TRUE*.

In these two situations, the output is set to the value of IN input.

ASC and DSC give the maximum ascendant and descendant growth during the TB time base.

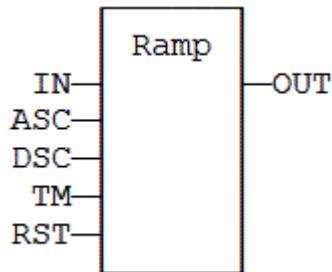
Both must be expressed as positive numbers.

In LD language, the operation is executed only if the input rung (EN) is *TRUE*. The output rung (ENO) keeps the same value as the input rung.

ST Language

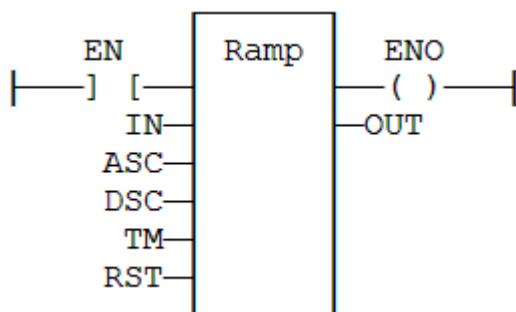
MyRamp is a declared instance of RAMP function block.

```
MyRamp (IN, ASC, DSC, TM, RST);  
OUT := MyRamp.OUT;
```

FBD Language**LD Language**

The function is executed only if EN is *TRUE*.

ENO keeps the same value as EN.

**IL Language**

MyRamp is a declared instance of RAMP function block.

Op1: CAL MyRamp (IN, ASC, DSC, TM, RST)

LD MyBlinker.OUT

ST OUT



2.31. Real Time Clock management functions

The following functions read the real time clock of the target system:

DTCurDate	Get current date stamp
DTCurTime	Get current time stamp
DTCurDateTime	Get current date and time stamp
DTDay	Get day from date stamp
DTMonth	Get month from date stamp
DTYear	Get year from date stamp
DTSec	Get seconds from time stamp
DTMin	Get minutes from time stamp
DTHour	Get hours from time stamp
DTMs	Get milliseconds from time stamp

The following functions format the current date/time to a string:

DAY_TIME	With predefined format
DTFORMAT	With custom format

The following functions are used for triggering operations:

DTAt	Pulse signal at the given date/time
DTEvery	Pulse signal with long period

Note:

Real Time clock may be not available on some targets.



2.31.1. Real Time Clock

Functions**DAY_TIME**

Get current date or time

Code `Q := DAY_TIME (SEL);`

SEL : DINT specifies the desired Information (see below)

Q : STRING desired Information formatted on a string

Possible values of SEL input:

	Value	Description
	1	current time - format: 'HH:MM:SS'
	2	day of the week
	0 (default)	current date - format: 'YYYY/MM/DD'
DTCURDATE		Get current date stamp
Code		<code>Q := DTCurDate ();</code>
	Q : DINT	numerical stamp representing the current date
DTCURTIME		Get current time stamp
Code		<code>Q := DTCurTime ();</code>
	Q : DINT	numerical stamp representing the current time of the day
DTCURDATETIME		Get current time stamp (function block)
Code		<code>Inst_DTCurDateTime (bLocal);</code>
	bLocal : BOOL	TRUE if local time is requested (GMT if FALSE).
	Year : DINT	Output: current year
	Month : DINT	Output: current month
	Day : DINT	Output: current day
	Hour : DINT	Output: current time: hours
	Min : DINT	Output: current time: minutes
	Sec : DINT	Output: current time: seconds
	MSec : DINT	Output: current time: milliseconds
	TmOfDay : TIME	Output: current time of day (since midnight)



DTYEAR	extract the year from a date stamp
Code	<code>Q := DTYear (iDate);</code>
IDATE : DINT	numerical stamp representing a date.
Q : DINT	year of the date (ex: 2004)
DTMONTH	extract the month from a date stamp
Code	<code>Q := DMonth (iDate);</code>
IDATE : DINT	numerical stamp representing a date
Q : DINT	month of the date (1..12)
DTDAY	extract the day of the month from a date stamp
Code	<code>Q := DDay (iDate);</code>
IDATE : DINT	numerical stamp representing a date
Q : DINT	day of the month of the date (1..31)
DTHOUR	extract the hours from a time stamp
Code	<code>Q := DTHour (iTime);</code>
ITIME : DINT	numerical stamp representing a time
Q : DINT	Hours of the time (0..23)
DTMIN	extract the minutes from a time stamp
Code	<code>Q := DTMin (iTime);</code>
ITIME : DINT	numerical stamp representing a time
Q : DINT	Minutes of the time (0..59)
DTSEC	extract the seconds from a time stamp
Code	<code>Q := DTSec (iTime);</code>
ITIME : DINT	numerical stamp representing a time
Q : DINT	Seconds of the time (0..59)
DTMS	extract the milliseconds from a time stamp
Code	<code>Q := DTMs (iTime);</code>
ITIME : DINT	numerical stamp representing a time.
Q : DINT	Milliseconds of the time (0..999)

2.31.2. DAY_TIME

Function Format the current date/time to a string

Inputs SEL : DINT Format selector

Outputs Q : STRING String containing formatted date or time

Note:

Real Time clock may be not available on some targets. Please refer to OEM instructions for further details about available features.

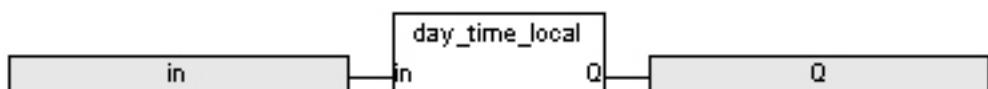
Remarks Possible values of the SEL input are:

Value	Meaning
1	current time - format: 'HH:MM:SS'.
2	day of the week.
0 (default)	current date - format: 'YYYY/MM/DD'.

In LD language, the operation is executed only if the input rung (EN) is *TRUE*. The output rung (ENO) keeps the same value as the input rung.

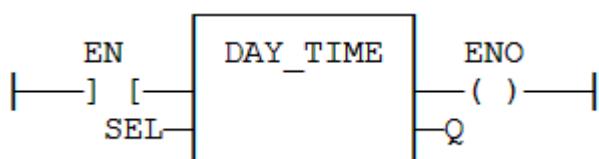
ST Language `Q := DAY_TIME (SEL);`

FBD Language



LD Language The function is executed only if EN is *TRUE*.

ENO keeps the same value as EN.



IL Language Op1: LD SEL

DAY_TIME

ST Q



2.31.3. DTFORMAT

Function Format the current date/time to a string with a custom format

Inputs FMT : STRING Format string

Outputs Q : STRING String containing formatted date or time

Note:

Real Time clock may be not available on some targets. Please refer to OEM instructions for further details about available features.

Remarks The format string may contain any character. Some special markers beginning with the '%' character indicates a date/time Information:

%Y Year including century (e.g. 2006)

%y Year without century (e.g. 06)

%m Month (1..12)

%d Day of the month (1..31)

%H Hours (0..23)

%M Minutes (0..59)

%S Seconds (0..59)

Example:

Example:

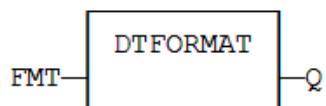
```
(* let's say we are at July 04th 2006, 18:45:20 *)
```

```
Q := DTFORMAT ('Today is %Y/%m/%d - %H:%M:%S');
```

```
(* Q is 'Today is 2006/07/04 - 18:45:20' *)
```

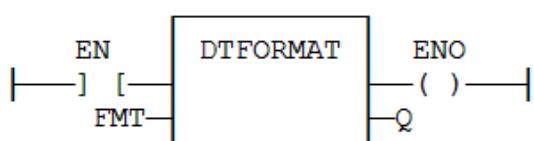
ST Language Q := DTFORMAT (FMT);

FBD Language



LD Language The function is executed only if EN is TRUE.

ENO keeps the same value as EN.



IL Language Op1: LD FMT

DTFORMAT

ST Q



2.31.4. DTAT

Function	Generate a pulse at given date and time	
Inputs	YEAR : DINT	Desired year (e.g. 2006)
	MONTH : DINT	Desired month (1 = January)
	DAY : DINT	Desired day (1 to 31)
	TMOFDAY : TIME	Desired time
Outputs	RST : BOOL	Reset command
	QAT : BOOL	Pulse signal
	QPAST : BOOL	<i>TRUE</i> if elapsed

Note:

Real Time clock may be not available on some targets. Please refer to OEM instructions for further details about available features.

Remarks Parameters are not updated constantly. They are taken into account when only:

- the first time the block is called.
- when the reset input (RST) is *TRUE*.

In these two situations, the outputs are reset to *FALSE*.

The first time the block is called with RST=*FALSE* and the specified date/stamp is passed, the output QPAST is set to *TRUE*, and the output QAT is set to *TRUE* for one cycle only (pulse signal).

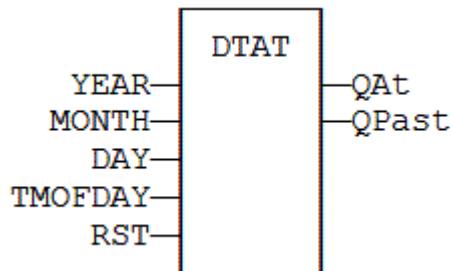
Highest units are ignored if set to 0. For instance, if arguments are *year=0, month=0, day = 3, tmofday=t#10h* then the block will trigger on the next 3rd day of the month at 10h.

In LD language, the block is activated only if the input rung is *TRUE*.

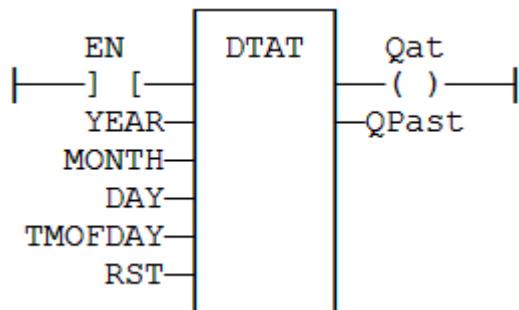
ST Language MyDTAT is a declared instance of DTAT function block.

```
MyDTAT(YEAR, MONTH, DAY, TMOFDAY, RST);  
QAT := MyDTAT.QAT;  
QPAST := MyDTATA.QPAST;
```

FBD Language



LD Language Called only if EN is *TRUE*.



IL Language MyDTAT is a declared instance of DTAT function block.

```
Op1: CAL MyDTAT (YEAR, MONTH, DAY, TMOFDAY, RST )
      LD MyDTAT.QAT
      ST QAT
      LD MyDTATA.QPAST
      ST QPAST
```

2.31.5. DTEVERY

Function	Generate a pulse signal with long period	
Inputs	RUN : DINT	Enabling command
	DAY : DINT	Period : number of days
	TM : TIME	Rest of the period (if not a multiple of 24h)
Outputs	Q : BOOL	Pulse signal
Remarks	This block provides a pulse signal with a period of more than 24h. The period is expressed as:	

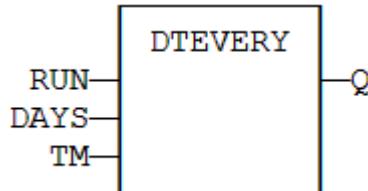
$$\text{DAY} * 24\text{h} + \text{TM}$$

For instance, specifying DAY=1 and TM=6h means a period of 30 hours.

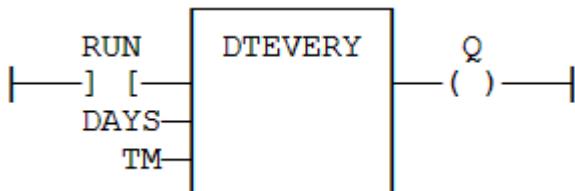
ST Language MyDTEVERY is a declared instance of DTEVERY function block.

```
MyDTEVERY (RUN DAYS, TM );  
Q := MyDTEVERY.Q;
```

FBD Language



LD Language



IL Language MyDTEVERY is a declared instance of DTEVERY function block.

```
Op1: CAL MyDTEVERY (RUN DAYS, TM )  
LD MyDTEVERY.Q  
ST Q
```



2.31.6. SerializeIn

Function	Extract the value of a variable from a binary frame	
Inputs	FRAME : USINT	Source buffer - must be an array
	DATA : ANY	Destination variable to be copied. DATA cannot be a STRING
	POS : DINT	Position in the source buffer
	BIGENDIAN : BOOL	TRUE if the frame is encoded with Big Endian format
Outputs	NEXTPOS : DINT	Position in the source buffer after the extracted data. 0 in case of error (invalid position / buffer size).

Remarks This function is commonly used for extracting data from a communication frame in binary format.

In LD language, the operation is executed only if the input rung (EN) is TRUE. The output rung (ENO) keeps the same value as the input rung. This function is not available in IL language.

The FRAME input must fit the input position and data size. If the value cannot be safely extracted, the function returns 0.

The DATA input must be directly connected to a variable, and cannot be a constant or complex expression. This variable will be forced with the extracted value.

The function extracts the following number of bytes from the source frame:

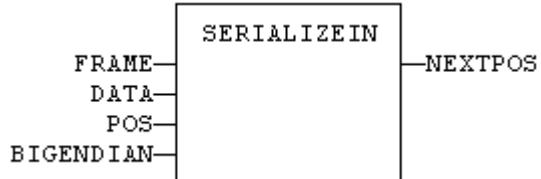
- 1 byte for BOOL, SINT, USINT and BYTE variables
- 2 bytes for INT, UINT and WORD variables
- 4 bytes for DINT, UDINT, DWORD and REAL variables
- 8 bytes for LINT and LREAL variables

The function cannot be used to serialize STRING variables.

The function returns the position in the source frame, after the extracted data. Thus the return value can be used as a position for the next serialization.

ST Language Q := SERIALIZEIN (FRAME, DATA, POS, BIGENDIAN);

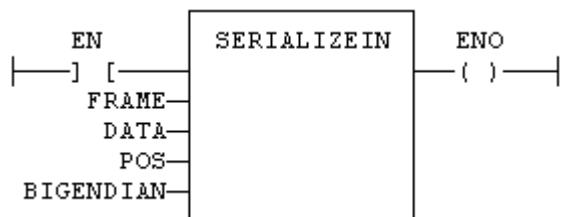
FBD Language





LD Language The function is executed only if EN is *TRUE*.

ENO keeps the same value as EN.



IL Language Not available.



2.31.7. SerializeOut

Function	Copy the value of a variable to a binary frame	
Inputs	FRAME : USINT	Destination buffer - must be an array
	DATA : ANY	Source variable to be copied. DATA cannot be a STRING
	POS : DINT	Position in the destination buffer
	BIGENDIAN : BOOL	TRUE if the frame is encoded with Big Endian format
Outputs	NEXTPOS : DINT	Position in the destination buffer after the copied data 0 in case of error (invalid position / buffer size)

Remarks This function is commonly used for building a communication frame in binary format.

In LD language, the operation is executed only if the input rung (EN) is TRUE. The output rung (ENO) keeps the same value as the input rung. This function is not available in IL language.

The FRAME input must be an array large enough to receive the data. If the data cannot be safely copied to the destination buffer, the function returns 0.

The function copies the following number of bytes to the destination frame:

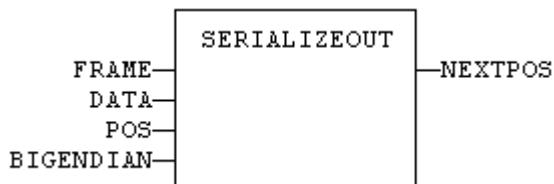
- 1 byte for BOOL, SINT, USINT and BYTE variables
- 2 bytes for INT, UINT and WORD variables
- 4 bytes for DINT, UDINT, DWORD and REAL variables
- 8 bytes for LINT and LREAL variables

The function cannot be used to serialize STRING variables.

The function returns the position in the destination frame, after the copied data. Thus the return value can be used as a position for the next serialization.

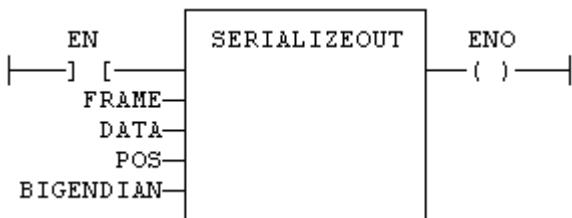
ST Language Q := SERIALIZEOUT (FRAME, DATA, POS, BIGENDIAN);

FBD Language



LD Language The function is executed only if EN is TRUE.

ENO keeps the same value as EN.



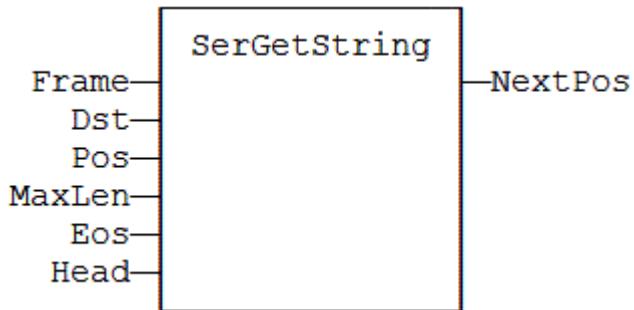
IL Language Not available.





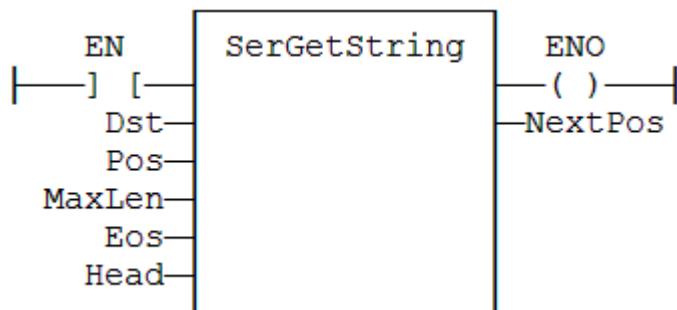
2.31.8. SerGetString

Function	Extract a string from a binary frame		
Inputs	FRAME : USINT	Source buffer - must be an array	
	DST : STRING	Destination variable to be copied	
	POS : DINT	Position in the source buffer	
	MAXLEN : DINT	Specifies a fixed length string	
	EOS : BOOL	Specifies a null terminated string	
	HEAD : BOOL	Specifies a string headed with its length	
Outputs	NEXTPOS : DINT	Position in the source buffer after the extracted data	
		0 in case of error (invalid position / buffer size)	
Remarks	<p>This function is commonly used for extracting data from a communication frame in binary format.</p> <p>In LD language, the operation is executed only if the input rung (EN) is <i>TRUE</i>. The output rung (ENO) keeps the same value as the input rung. This function is not available in IL language.</p> <p>The FRAME input must fit the input position and data size. If the value cannot be safely extracted, the function returns 0.</p> <p>The DST input must be directly connected to a variable, and cannot be a constant or complex expression. This variable will be forced with the extracted value.</p> <p>The function extracts the following bytes from the source frame:</p>		
MAXLEN	EOS	HEAD	Description
<i><> 0</i>	<i>any</i>	<i>any</i>	The string is stored on a fixed length specified by MAXLEN. If the string is actually smaller, the space is completed with null bytes.
<i>= 0</i>	<i>TRUE</i>	<i>any</i>	The string is stored with its actual length and terminated by a null byte.
<i>= 0</i>	<i>FALSE</i>	<i>TRUE</i>	The string is stored with its actual length and preceded by its length stored on one byte
<i>= 0</i>	<i>FALSE</i>	<i>FALSE</i>	invalid call
The function returns the position in the source frame, after the extracted data. Thus the return value can be used as a position for the next serialization.			
ST Language	<code>Q := SerGetString (FRAME, DSR, POS, MAXLEN, EOS, HEAD);</code>		

FBD Language**LD Language**

The function is executed only if EN is *TRUE*.

ENO keeps the same value as EN.

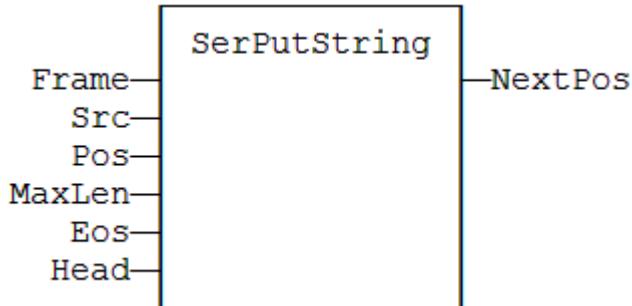
**IL Language**

Not available.



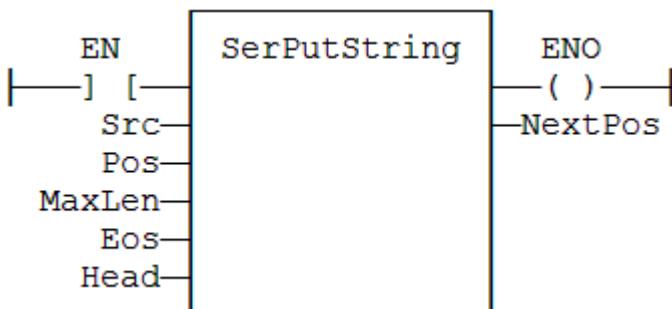
2.31.9. SerPutString

Function	Copies a string to a binary frame		
Inputs	FRAME : USINT	Destination buffer - must be an array	
	DST : STRING	Source variable to be copied	
	POS : DINT	Position in the source buffer	
	MAXLEN : DINT	Specifies a fixed length string	
	EOS : BOOL	Specifies a null terminated string	
	HEAD : BOOL	Specifies a string headed with its length	
Outputs	NEXTPOS : DINT	Position in the destination buffer after the copied data	
		0 in case of error (invalid position / buffer size)	
Remarks	<p>This function is commonly used for storing data to a communication frame.</p> <p>In LD language, the operation is executed only if the input rung (EN) is <i>TRUE</i>. The output rung (ENO) keeps the same value as the input rung. This function is not available in IL language.</p> <p>The FRAME input must fit the input position and data size. If the value cannot be safely copied, the function returns 0.</p> <p>The function copies the following bytes to the frame:</p>		
MAXLEN	EOS	HEAD	Description
<> 0	any	any	The string is stored on a fixed length specified by MAXLEN. If the string is actually smaller, the space is completed with null bytes. If the string is longer, it is truncated.
= 0	TRUE	any	The string is stored with its actual length and terminated by anull byte.
= 0	FALSE	TRUE	The string is stored with its actual length and preceded by its length stored on one byte.
=0	FALSE	FALSE	invalid call
The function returns the position in the source frame, after the stored data. Thus the return value can be used as a position for the next serialization.			
ST Language	<code>Q := SerPutString (FRAME, DSR, POS, MAXLEN, EOS, HEAD);</code>		

FBD Language**LD Language**

The function is executed only if EN is *TRUE*.

ENO keeps the same value as EN.

**IL Language**

Not available.



2.31.10. SERIO

Function Block Serial communication

Inputs RUN : BOOL Enable communication (opens the comm port)

SND : BOOL *TRUE* if data has to be sent

CONF : STRING Configuration of the communication port

DATASND : STRING Data to send

Outputs OPEN : BOOL *TRUE* if the communication port is open

RCV : BOOL *TRUE* if data has been received

ERR : BOOL *TRUE* if error detected during sending data

DATARCV : STRING Received data

Remarks The RUN input does not include an edge detection. The block tries to open the port on each call if RUN is *TRUE* and if the port is still not successfully open. The CONF input is used for settings when opening the port. Please refer to your OEM instructions for further details about possible parameters.

The SND input does not include an edge detection. Characters are sent on each call if SND is *TRUE* and DATASND is not empty.

The DATARCV string is erased on each cycle with received data (if any). Your application is responsible for checking or storing received character immediately after the call to SERIO block.

SERIO is available during simulation. In that case, the CONF input defines the communication port according to the syntax of the MODE command. For Example:

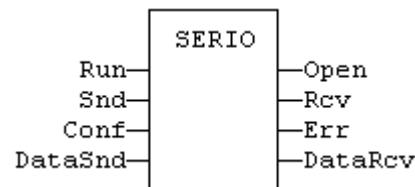
'COM1:9600,N,8,1'

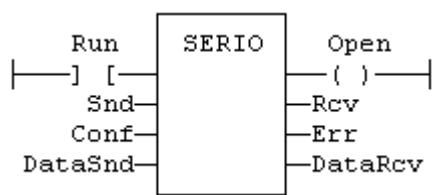
The SERIO block may not be supported on some targets. Refer to your OEM instructions for further details.

ST Language MySer is a declared instance of SERIO function block.

```
MySer (RUN, SND, CONF, DATASND );
OPEN := MySer.OPEN;
RCV := MySer.RCV;
ERR := MySer.ERR;
DATARCV := MySer.DATARCV;
```

FBD Language



LD Language

IL Language MySer is a declared instance of SERIO function block.

Op1: CAL MySer (RUN, SND, CONF, DATASND)

```
LD  MySer.OPEN
ST  OPEN
LD  MySer.RCV
ST  RCV
LD  MySer.ERR
ST  ERR
LD  MySer.DATARCV
ST  DATARCV
```

2.31.11. SigID

Function	Get the identifier of a Signal resource	
Inputs	SIGNAL : STRING	Name of the signal resource - must be a constant value!
	COL : STRING	Name of the column within the signal resource - must be a constant value!
Outputs	ID : DINT	ID of the signal - to be passed to other blocks
Remarks	Some blocks have arguments that refer to a <i>signal</i> resource. For all these blocks, the signal argument is materialized by a numerical identifier. This function enables you to get the identifier of a signal defined as a resource.	
ST Language	<pre>ID := SigID ('MySignal', 'FirstColumn');</pre>	
FBD Language		
LD Language		
IL Language	<pre>Op1: LD 'MySignal' SigID 'FirstColumn' ST ID</pre>	

2.31.12. SigPlay

Function	Generate a signal defined in a resource	
Inputs	IN : BOOL	Triggering command
	ID : DINT	ID of the signal resource, provided by SigID function
	RST : BOOL	Reset command
	TM : TIME	Minimum duration in between two changes of the output
Outputs	Q : BOOL	<i>TRUE</i> when the signal is finished
	OUT : REAL	Generated signal
	ET : TIME	Elapsed time
Remarks	<p>The <i>ID</i> argument is the identifier of the <i>signal</i> resource. Use the SigID function to get this value.</p> <p>The <i>IN</i> argument is used as a Play / Pause command to play the signal. The signal is not reset to the beginning when <i>IN</i> becomes <i>FALSE</i>. Instead, use the <i>RST</i> input that resets the signal and forces the <i>OUT</i> output to <i>0</i>.</p> <p>The <i>TM</i> input specifies the minimum amount of time in between two changes of the output signal. This parameter is ignored if less than the cycle scan time.</p> <p>This function block includes its own timer. Alternatively, you can use the SigScale function if you want to trigger the signal using a specific timer.</p>	
ST Language	MySig is a declared instance of SIGPLAY function block. <code>MySig (IN, ID, RST, TM); Q := MySig.Q; OUT := MySig.OUT; ET := MySig.ET;</code>	
FBD Language		
LD Language		
IL Language	MySig is a declared instance of SIGPLAY function block. <code>Op1: CAL MySig (IN, ID, RST, TM) LD MySig.Q</code>	



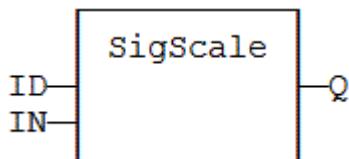
```
ST Q
LD MySig.OUT
ST OUT
LD MySig.ET
ST ET
```

2.31.13. SigScale

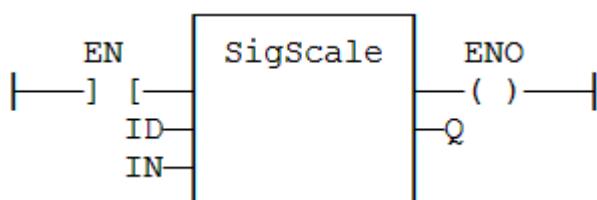
Function	Get a point from a Signal resource	
Inputs	ID : DINT	ID of the signal resource, provided by SigID function
	IN : TIME	Time (X) coordinate of the desired point within the signal resource
Outputs	Q : REAL	Value (Y) coordinate of the point in the signal
Remarks	The <i>ID</i> argument is the identifier of the <i>signal</i> resource. Use the SigID function to get this value.	
	This function converts a time value to a analog value such as defined in the signal resource. This function can be used instead of SigPlay function block if you want to trigger the signal using a specific timer.	

ST Language `Q := SigScale (ID, IN);`

FBD Language



LD Language



IL Language `Op1: LD IN`

 SigScale ID

 ST Q

2.31.14. STACKINT

Function	Manages a stack of DINT integers	
Inputs	PUSH : BOOL	Command: when changing from <i>FALSE</i> to <i>TRUE</i> , the value of IN is pushed on the stack
	POP : BOOL	Pop command: when changing from <i>FALSE</i> to <i>TRUE</i> , deletes the top of the stack
	R1 : BOOL	Reset command: if <i>TRUE</i> , the stack is emptied and its size is set to N
	IN : DINT	Value to be pushed on a rising pulse of PUSH
	N : DINT	maximum stack size - cannot exceed 128
Outputs	EMPTY : BOOL	<i>TRUE</i> if the stack is empty
	OFLO : BOOL	<i>TRUE</i> if the stack is full
	OUT : DINT	value at the top of the stack
Remarks	Push and pop operations are performed on rising pulse of PUSH and POP inputs. In LD language, the input rung is the PUSH command. The output rung is the EMPTY output. The specified size (N) is taken into account only when the R1 (reset) input is <i>TRUE</i> .	
ST Language	MyStack is a declared instance of STACKINT function block. <code>MyStack (PUSH, POP, R1, IN, N); EMPTY := MyStack.EMPTY; OFLO := MyStack.OFLO; OUT := MyStack.OUT;</code>	
FBD Language		
LD Language		
IL Language	MyStack is a declared instance of STACKINT function block. <code>Op1: CAL MyStack (PUSH, POP, R1, IN, N) LD MyStack.EMPTY ST EMPTY LD MyStack.OFLO</code>	



ST OFLO
LD MyStack.OUT
ST OUT



2.31.15. SurfLin

Function Block Linear interpolation on a surface

Inputs	X : REAL	X coordinate of the point to be interpolated
	Y : REAL	Y coordinate of the point to be interpolated
	XAxis : REAL[]	X coordinates of the known points of the X axis
	YAxis : REAL[]	Y coordinates of the known points of the Y axis
	ZVal : REAL[,]	Z coordinate of the points defined by the axis

Outputs	Z : REAL	Interpolated Z value corresponding to the X,Y input point
	OK : BOOL	<i>TRUE</i> if successful
	ERR : DINT	Error code if failed - 0 if OK

Remarks This function performs linear surface interpolation in between a list of points defined in XAxis and YAxis single dimension arrays. The output Z value is an interpolation of the Z values of the four rounding points defined in the axis. Z values of defined points are passed in the ZVal matrix (two dimension array).

ZVal dimensions must be understood as: ZVal [iX , iY]

Values in X and Y axis must be sorted from the smallest to the biggest. There must be at least two points defined in each axis. ZVal must fit the dimension of XAxis and YAxis arrays. For instance:

```
XAxis : ARRAY [0..2] of REAL;  
YAxis : ARRAY [0..3] of REAL;  
ZVal : ARRAY [0..2,0..3] of REAL;
```

In case the input point is outside the rectangle defined by XAxis and YAxis limits, the Z output is bound to the corresponding value and an error is reported.

The ERR output gives the cause of the error if the function fails:

Error code	Meaning
0	OK
1	Invalid dimension of input arrays
2	Invalid points for the X axis
3	Invalid points for the Y axis
4	X,Y point is out of the defined axis



2.32. TCP/IP Management

The following functions enable management of TCP-IP sockets for building client or server applications over ETHERNET network:

tcpListen	create a listening server socket
tcpAccept	accept client connections
tcpConnect	create a client socket and connect it to a server
tcpIsConnected	test if a client socket is connected
tcpClose	close a socket
tcpSend	send characters
tcpReceive	receive characters
tcpIsValid	test if a socket is valid

Each socket is identified in the application by a unique handle manipulated as a DINT value.



Attention

- Although the system provides a simplified interface, you must be familiar with the socket interface such as existing in other programming languages such as "C".
- Socket management may be not available on some targets. Please refer to OEM instructions for further details about available features.



2.32.1. TCP/IP Management Functions

Functions	tcpListen	Create a listening server socket
	Code	SOCK := tcpListen (PORT, MAXCNX);
	PORT : DINT	TCP port number to be attached to the server socket
	MAXCNX : DINT	maximum number of client sockets that can be accepted
	SOCK : DINT	ID of the new server socket
Description	This function creates a new socket performs the <i>bind</i> and <i>listen</i> operations using default TCP settings. You will have to call the <i>tcpClose</i> function to release the socket returned by this function	
<hr/>		
DTCURDATE	Get current date stamp	
	Code	Q := DTCurDate();
	Q : DINT	numerical stamp representing the current date
<hr/>		
DTCURTIME	Get current time stamp	
	Code	Q := DTCurTime();
	Q : DINT	numerical stamp representing the current time of the day
<hr/>		
DTCURDATETIME	Get current time stamp (function block)	
	Code	Inst_DTCurDateTime (bLocal);
	bLocal : BOOL	TRUE if local time is requested (GMT if FALSE).
	Year : DINT	Output: current year
	Month : DINT	Output: current month
	Day : DINT	Output: current day
	Hour : DINT	Output: current time: hours
	Min : DINT	Output: current time: minutes
	Sec : DINT	Output: current time: seconds
	MSec : DINT	Output: current time: milliseconds



	TmOfDay : TIME	Output: current time of day (since midnight)
DTYEAR	extract the year from a date stamp	
Code	<code>Q := DTYear (iDate);</code>	
	IDATE : DINT	numerical stamp representing a date.
	Q : DINT	year of the date (ex: 2004)
DTMONTH	extract the month from a date stamp	
Code	<code>Q := DMonth (iDate);</code>	
	IDATE : DINT	numerical stamp representing a date
	Q : DINT	month of the date (1..12)
DTDAY	extract the day of the month from a date stamp	
Code	<code>Q := DDay (iDate);</code>	
	IDATE : DINT	numerical stamp representing a date
	Q : DINT	day of the month of the date (1..31)
DTHOUR	extract the hours from a time stamp	
Code	<code>Q := DTHour (iTime);</code>	
	ITIME : DINT	numerical stamp representing a time
	Q : DINT	Hours of the time (0..23)
DTMIN	extract the minutes from a time stamp	
Code	<code>Q := DTMin (iTime);</code>	
	ITIME : DINT	numerical stamp representing a time
	Q : DINT	Minutes of the time (0..59)
DTSEC	extract the seconds from a time stamp	
Code	<code>Q := DTSec (iTime);</code>	
	ITIME : DINT	numerical stamp representing a time



Q : DINT Seconds of the time (0..59)

DTMS extract the milliseconds from a time stamp

Code `Q := DTMs (iTime);`

iTIME : DINT numerical stamp representing a time.

Q : DINT Milliseconds of the time (0..999)

tcpAccept accept a new client connection

Code `SOCK := tcpAccept (LSOCK);`

LSOCK : DINT ID of a server socket returned by the `tcpListen` function

SOCK : DINT ID of a new client socket accepted, or invalid ID if no new connection

Description This functions performs the *accept* operation using default TCP settings. You will have to call the `tcpClose` function to release the socket returned by this function.

tcpConnect create a client socket and connect it to a server

Code `SOCK := tcpConnect (ADDRESS, PORT);`

ADDRESS : STRING IP address of the remote server

PORT : DINT desired port number on the server

SOCK : DINT ID of the new client socket

Description This functions creates a new socket performs the *connect* operation using default TCP settings and specified server address and port. You will have to call the `tcpClose` function to release the socket returned by this function.

Note:

It is possible that the functions returns a valid socket ID even if the connection to the server is not yet actually performed. After calling this function, you must call `tcpIsConnected` function to know if the connection is ready.



tcpIsConnected	test if a client socket is connected
Code	<code>OK := tcpIsConnected (SOCK);</code>
SOCK : DINT	ID of the client socket
OK : BOOL	<i>TRUE</i> if connection is correctly established

 **Attention**

It is possible that the socket becomes invalid after this function is called, if an error occurs in the TCP connection. You must call the `tcpIsValid` function after calling this function. If the socket is not valid anymore then you must close it by calling `tcpClose`.

tcpClose	release a socket
Code	<code>OK := tcpClose (SOCK);</code>
SOCK : DINT	ID of any socket
OK : BOOL	<i>TRUE</i> if successful

Description You are responsible for closing any socket created by `tcpListen`, `tcpAccept` or `tcpConnect` functions, even if they have become invalid.

tcpSend	send characters
Code	<code>NBSENT := tcpSend (SOCK, NBCHR, DATA);</code>
SOCK : DINT	ID of a socket
NBCHR : DINT	number of characters to be sent
DATA : STRING	string containing characters to send
NBSENT : DINT	number of characters actually sent

Description It is possible that the number of characters actually sent is less than the number expected. In that case, you will have to call again the function on the next cycle to send the pending characters.

 **Attention**

It is possible that the socket becomes invalid after this function is called, if an error occurs in the TCP connection. You must call the `tcpIsValid` function after calling this function. If the socket is not valid anymore then you must close it by calling `tcpClose`.



tcpReceive	receive characters	
Code	<code>NBRCV := tcpReceive (SOCK, MAXCHR, DATA);</code>	
SOCK : DINT	ID of a socket	
MAXCHR : DINT	maximum number of characters desired	
DATA : STRING	string where to store received characters	
NBRCV : DINT	number of characters actually received	

Description It is possible that the number of characters actually received is less than the number expected. In that case, you will have to call again the function on the next cycle to receive the pending characters.

 **Attention**

It is possible that the socket becomes invalid after this function is called, if an error occurs in the TCP connection. You must call the `tcpIsValid` function after calling this function. If the socket is not valid anymore then you must close it by calling `tcpClose`.

tcpIsValid	test if a socket is valid	
Code	<code>OK := tcpIsValid (SOCK);</code>	
SOCK : DINT	ID of the socket	
OK : BOOL	<i>TRUE</i> if specified socket is still valid	



2.33. Text buffers manipulation

Strings are limited to 255 characters. The following is a set of functions and function blocks for working with non limited text buffers. Text buffers are dynamically allocated or re-allocated.

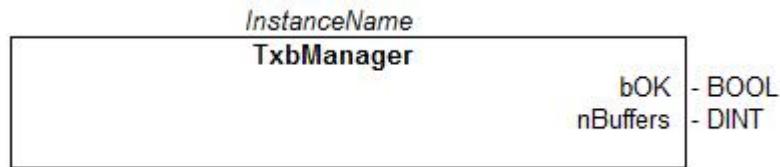
Attention

- Text buffers management functions use safe dynamic memory allocation that needs to be configured in the project settings.
From the project settings, press **Advanced** button and select the **Memory** tab. Here you can setup the memory for safe dynamic allocation.
- There must be one instance of the TxbManager declared in your application when using these functions.
- The application should take care to release memory allocated for each buffer. Allocating buffers without freing them will lead to memory leaks.

The application is responsible for freeing all allocated text buffers. However, all allocated buffers are automatically released when the application stops.

2.33.1. Memory management / Miscellaneous

Functions TxbManager Text buffer manager



Output bOK : BOOL *TRUE* if the text buffers memory system is correctly initialized.

nBuffers : DINT Number of text buffers currently allocated in memory.

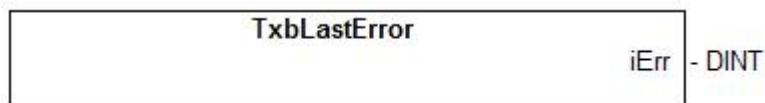
Description This function block is used for managing the memory allocated for text buffers. It takes care of releasing the corresponding memory when the application stops, and can be used for tracking memory leaks.



Attention

There must be one and only one instance of this block declared in the IEC application in order to use any other Txb... function.

TxbLastError Text buffer last error



Output iErr : DINT Error code reported by the last call:

0 = OK

other = error (see below)

Below are possible error codes:

Code	Meaning
1	Invalid instance of TXBManager - should be only one
2	Manager already open - should be only one instance of TxbManager
3	Manager not open - no instance of TxbManager declared
4	Invalid handle
5	String has been truncated during copy
6	Cannot read file

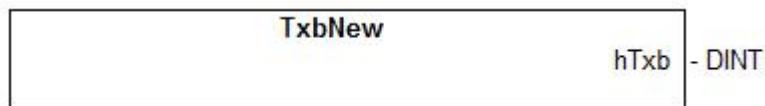


- 7 Cannot write file
- 8 Unsupported data type
- 9 Too many text buffers allocated

Description All TXB functions and blocks simply return a Boolean Information as a return value. This function can be called after any other function giving a FALSE return. It gives a detailed error code about the last detected error.

2.33.2. Allocation / exchange with files

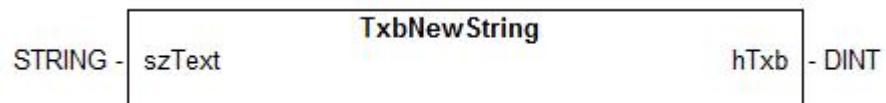
Functions TxbNew New Text buffer



Output hTxb : DINT Handle of the new buffer

Description This function allocates a new text buffer initially empty. The application will be responsible for releasing the buffer by calling the TxbFree function.

TxbNewString Text buffer new string



Input szText : STRING Initial value of the text buffer

Output hTxb : DINT Handle of the new buffer

Description This function allocates a new text buffer initially filled with the specified string. The application will be responsible for releasing the buffer by calling the TxbFree function

TxbFree Text buffer free



Input hTxb : DINT Handle of a valid text buffer

Output bOK : BOOL *TRUE* if successful

Description This function releases a text buffer from memory.



TxbReadFile Text buffer read file

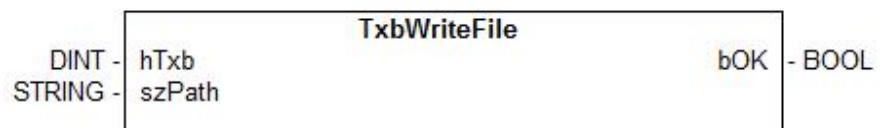


Input szPath : STRING Full qualified pathname of the file to be read

Output hTxb : DINT Handle of the new buffer

Description This function allocates a new text buffer and fills it with the contents of the specified file. The application will be responsible for releasing the buffer by calling the TxbFree function

TxbWriteFile Text buffer write file



Input hTxb : DINT Handle of the text buffer

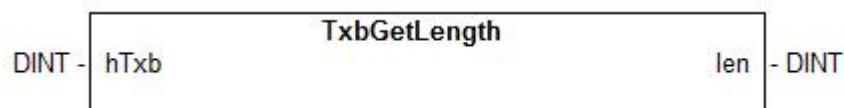
szPath : STRING Full qualified pathname of the file to be created

Output bOK : BOOL *TRUE* if successful

Description This function stores the contents of a text buffer to a file. The text buffer remains allocated in memory

2.33.3. Data exchange

Functions **TxbGetLength** Get length of the text buffer

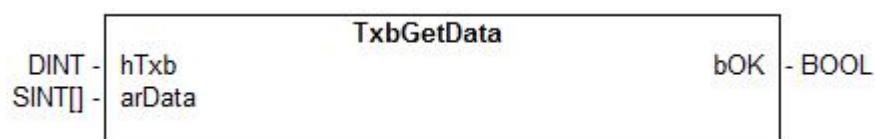


Input hTxb : DINT Handle of the text buffer

Output len : DINT Number of characters in the text buffer

Description This function returns the current length of a text buffer

TxbGetData Get data of the text buffer



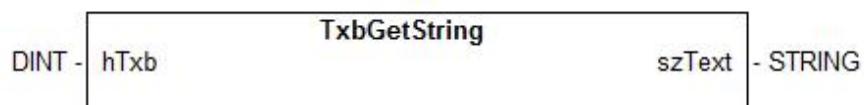
Input hTxb : DINT Handle of the text buffer

arData : SINT[] Array of characters to be filled with text

Output bOK : BOOL *TRUE* if successful.

Description This function copies the contents of a text buffer to an array of characters.

TxbGetString Get string of the text buffer



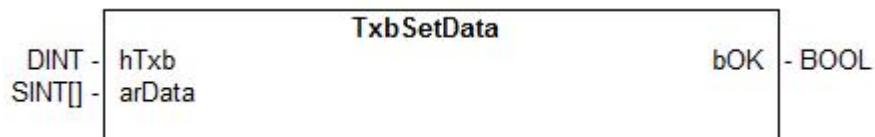
Input hTxb : DINT Handle of the text buffer

Output szText : STRING String to be filled with text

Description This function copies the contents of a text buffer to a string. The text is truncated if the string is not large enough.



TxbSetData Set the data of the text buffer



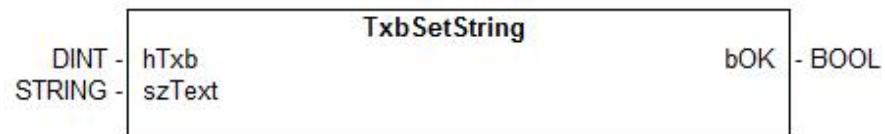
Input **hTxb : DINT** Handle of the text buffer

arData : SINT[] Array of characters to copy

Output **bOK : BOOL** *TRUE* if successful

Description This function copies an array of characters to a text buffer. All characters of the input array are copied.

TxbSetString Set the string of the text buffer



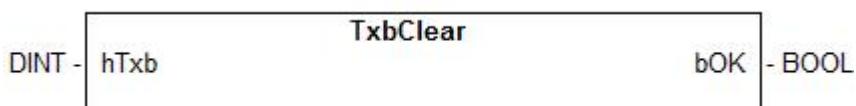
Input **hTxb : DINT** Handle of the text buffer

szText : STRING String to be copied

Output **bOK : BOOL** *TRUE* if successful

Description This function copies the contents of a string to a text buffer.

TxbClear Clears the text buffer



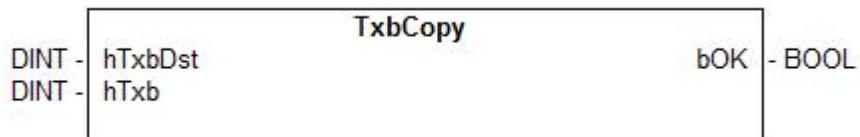
Input **hTxb : DINT** Handle of the text buffer

Output **bOK : BOOL** *TRUE* if successful

Description This function empties a text buffer.



TxbCopy Copies the text buffer



Input hTxbDst : DINT Handle of the destination text buffer.

hTxb : DINT Handle of the source text buffer.

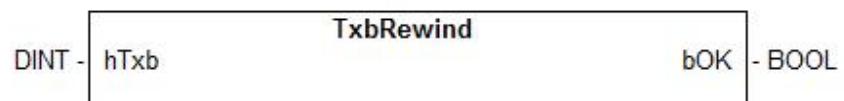
Output bOK : BOOL *TRUE* if successful.

Description This function copies the contents of the hTxb buffer to the hTxbDst buffer.



2.33.4. Sequential reading

TxbRewind Resets the text buffer

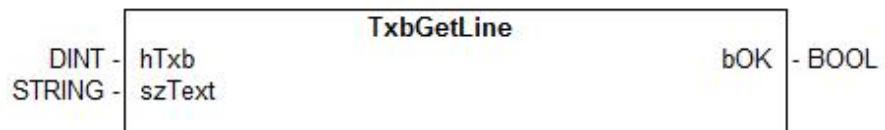


Input hTxb : DINT Handle of the text buffer.

Output bOK : BOOL *TRUE* if successful.

Description This function resets the sequential reading of a text buffer (rewind to the beginning of the text).

TxbGetLine Get the sequential line of the text buffer



Input hTxb : DINT Handle of the text buffer

szText : STRING String to be filled with read line

Output bOK : BOOL *TRUE* if successful

Description This function sequentially reads a line of text from a text buffer. End of line characters are not copied to the output string.

2.33.5. Sequential writing

TxbRewind Resets the text buffer



Input hTxb : DINT Handle of the text buffer.

Output bOK : BOOL *TRUE* if successful.

Description This function resets the sequential reading of a text buffer (rewind to the beginning of the text).

TxbGetLine Get the sequential line of the text buffer



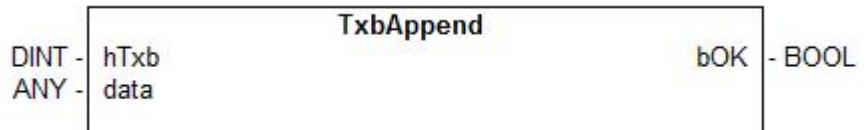
Input hTxb : DINT Handle of the text buffer

szText : STRING String to be filled with read line

Output bOK : BOOL *TRUE* if successful

Description This function sequentially reads a line of text from a text buffer. End of line characters are not copied to the output string.

TxbAppend Append to the text buffer



Input hTxb : DINT Handle of the text buffer

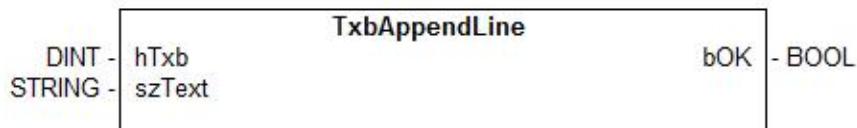
data : ANY Any variable

Output bOK : BOOL *TRUE* if successful

Description This function adds the contents of a variable, formatted as text, to a text buffer. The specified variable can have any data type.



TxbAppendLine Appends a line to the text buffer



Input hTxb : DINT Handle of the text buffer

szText : STRING String to be added to the text

Output bOK : BOOL *TRUE* if successful

Description This function adds the contents of the specified string variable to a text buffer, plus end of line characters.

TxbAppendEol Append end of line characters to the text buffer



Input hTxb : DINT Handle of the text buffer

Output bOK : BOOL *TRUE* if successful

Description This function adds end of line characters to a text buffer.

TxbAppendTxb Append end of another text buffer to the text buffer

Input hTxbDst : DINT Handle of the text buffer to be completed

hTxb : DINT Handle of the text buffer to be added

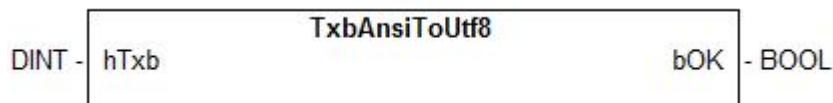
Output bOK : BOOL *TRUE* if successful

Description This function adds the contents of the *hTxb* text buffer to the *hTxbDst* text buffer.



2.33.6. UNICODE conversions

TxbAnsiToUtf8 Converts text buffer contents from ANSI to UTF8



Input hTxb : DINT Handle of the text buffer.

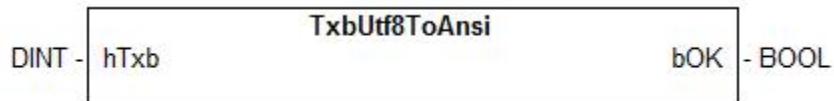
Output bOK : BOOL *TRUE* if successful.

Description This function converts the whole contents of a text buffer from ANSI to UNICODE UTF8 encoding. Warning:

Attention

- This function may be time and memory consuming for large buffers.
- UNICODE conversion may be not available on some operating systems

TxbUtf8ToAnsi Converts text buffer contents from UTF8 to ANSI



Input hTxb : DINT Handle of the text buffer.

Output bOK : BOOL *TRUE* if successful.

Description This function converts the whole contents of a text buffer from UNICODE UTF8 to ANSI encoding. Warning:

Attention

- This function may be time and memory consuming for large buffers.
- UNICODE conversion may be not available on some operating systems



2.34. UDP Management

The following functions enable management of UDP sockets for building client or server applications over ETHERNET network:

udpCreate	create a UDP socket
udpAddrMake	build an address buffer for UDP functions
updSend	To send a telegram
udpRcvFrom	receive a telegram
udpClose	close a socket
udpIsValid	test if a socket is valid.

Each socket is identified in the application by a unique handle manipulated as a DINT value.



Attention

- Although the system provides a simplified interface, you must be familiar with the socket interface such as existing in other programming languages such as "C".
- Socket management may be not available on some targets. Please refer to OEM instructions for further details about available features.



2.34.1. UDP Management Functions

udpCreate create a UDP socket

Code `SOCK := udpCreate (PORT);`

PORT : DINT TCP port number to be attached to the server socket or 0 for a client socket

SOCK : DINT ID of the new server socket

Description This functions creates a new UDP socket. If the PORT argument is *not* 0, the socket is bound to the port and thus can be used as a server socket.

udpAddrMake build an address buffer for UDP functions

Code `OK := udpAddrMake (IPADDR, PORT, ADD);`

IPADDR : STRING IP address in form xxx.xxx.xxx.xxx

PORT : DINT IP port number

ADD : USINT[32] Buffer where to store the UDP address (filled on output)

OK : BOOL *TRUE* if successful

Description This functions is required for building a internal *UDP* address to be passed to the *udpSendTo* function in case of *UDP* client processing.

udpSendTo send a UDP telegram

Code `OK := udpAddrMake (IPADDR, PORT, ADD);`

SOCK : DINT ID of the client socket

NB : DINT number of characters to send

ADD : USINT[32] buffer containing the UDP address (on input)

DATA : STRING characters to send

OK : BOOL *TRUE* if successful

Description The *ADD* buffer must contain a valid *UDP* address either constructed by the *udpAddrMake* function or returned by the *udpRcvFrom* function.

udpRcvFrom receive a UDP telegram

Code `OK := udpRcvFrom (SOCK, NB, ADD, DATA);`



SOCK : DINT	ID of the client socket
NB : DINT	Maximum number of characters received
ADD : USINT[32]	Buffer containing the UDP address of the transmitter (filled on output)
DATA : STRING	buffer where to store received characters
Q : DINT	number of actually received characters

Description If characters are received, the function fills the *ADD* argument with the internal *UDP* of the sender. This buffer can then be passed to the *udpSendTo* function to send the answer.

udpClose release a socket

Code `OK := udpClose (SOCK);`

SOCK : DINT ID of any socket

OK : BOOL *TRUE* if successful

Description Please close any socket created by *tcpListen*, *tcpAccept* or *tcpConnect* functions, even if they have become invalid.

udpIsValid test if a socket is valid

Code `OK := udpIsValid (SOCK);`

SOCK : DINT ID of the socket

OK : BOOL *TRUE* if specified socket is still valid

2.35. VLID

Function	Get the identifier of an embedded list of variables
Inputs	FILE : STRING Pathname of the list file (.SPL or .TXT) - must be a constant value!
Outputs	ID : DINT ID of the list - to be passed to other blocks
Remarks	Some blocks have arguments that refer to a list of variables. For all these blocks, the <i>list</i> argument is materialized by a numerical identifier. This function enables you to get the identifier of a list of variables.

Embedded lists of variables can be:

- Watch lists created with the Workbench. Such files are suffixed with .SPL.
- Simple .TXT text files with one variable name per line.

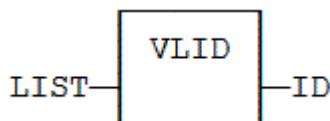
Lists must contain single variables only. Items of arrays and structures must be specified one by one. The length of the list is not limited by the system.

Note:

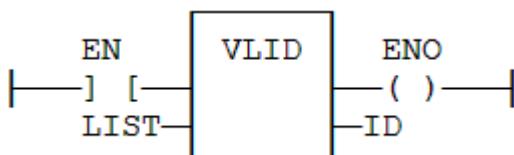
List files are read at compiling time and are embedded into the downloaded application code. This implies that a modification performed in the list file after downloading will not be taken into account by the application.

ST Language `ID := VLID ('MyFile.spl');`

FBD Language



LD Language The function is executed only if EN is *TRUE*.



IL Language `Op1: LD 'MyFile.spl'`

`VLID COL`

`ST ID`



2.36. XML writing and parsing

XML documents can be managed from the IEC application as a tree of XML tags. Each XML tag is represented by:

- a tag name
- a list of attributes (name=value)
- a contents (raw text)
- a list of child tags

A whole XML document is handled as its main tag.

XML functions are originally designed for working with text buffers (**Error! Bookmark not defined.**). However, some additional functions have been added for dealing with XML files on disk as it may be helpful in other applications.

Attention

XML parsing and writing functions support only 8 bit encoding features. This can be UTF8 or ANSI (ISO-8859-1) encoding. Some text buffers handling functions provide conversion between UTF8 and ANSI formats.

There must be one instance of the XmlManager declared in your application when using these functions.

There must be one instance of the TxbManager declared in your application when using text buffer related functions.

The following functions are used in XML writing and parsing.

Memory management and general

XmlManager (Error! Bookmark not defined.) Main gatherer of XML data in memory

XmlLastError (Error! Bookmark not defined.) Get detailed error report about last call

Reading/writing documents (Error! Bookmark not defined.)

XmlNewDoc (Error! Bookmark not defined.) Create an empty XML document

XmlFreeDoc (Error! Bookmark not defined.) Release an XML document from memory

XmlParseDocTxb (Error! Bookmark not defined.) Parse an XML document from text buffer

XmlParseDocFile (Error! Bookmark not defined.) Parse an XML document from file



XmlWriteDocTxb (Error! Bookmark not defined.) Format XML data to text buffer

XmlWriteDocFile (Error! Bookmark not defined.) Format XML data to file

Exploring an XML document (Error! Bookmark not defined.)

XmlGetTagName (Error! Bookmark not defined.) Get name of an XML tag

XmlCheckTagName (Error! Bookmark not defined.) Test the name of an XML tag

XmlGetTagCont (Error! Bookmark not defined.) Get tag contents as STRING

XmlGetTagContT (Error! Bookmark not defined.) Get tag contents as text buffer

XmlGetTagAttrib (Error! Bookmark not defined.) Get tag attribute

XmlGetParent (Error! Bookmark not defined.) Get parent tag

XmlFirstChild (Error! Bookmark not defined.) Start enumerating child tags

XmlNextChild (Error! Bookmark not defined.) Enumerate child tags

Building an XML document (Error! Bookmark not defined.)

XmlSetTagAttrib (Error! Bookmark not defined.) Set tag attribute

XmlSetTagCont (Error! Bookmark not defined.) Set tag contents with STRING

XmlSetTagContT (Error! Bookmark not defined.) Set tag contents with text buffer

XmlAddChild (Error! Bookmark not defined.) Add a child tag

Exchanging variables using their name (Error! Bookmark not defined.)

XmlGetSybValue (Error! Bookmark not defined.) Get variable value using its symbol

XmlSetSybValue (Error!) Set variable value using its symbol



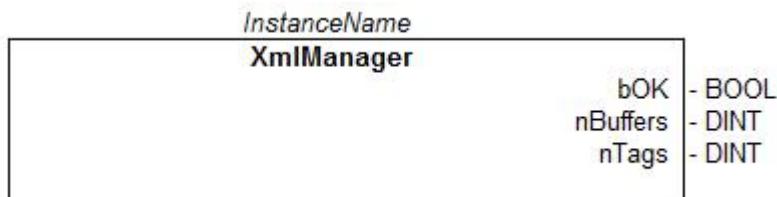
Bookmark not defined.)



2.36.1. XmlManager

Function block Get the identifier of an embedded list of variables

Diagram



Outputs

bOK : BOOL	TRUE if the XML memory system is correctly initialized
nBuffers : DINT	Number of XML documents currently allocated in memory
nTags : DINT	Number of XML tags currently allocated in memory

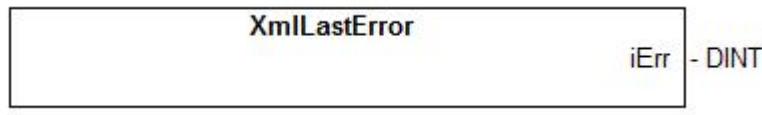
Description

This function block is used for managing the memory allocated for XML tags and documents. It takes care of releasing the corresponding memory when the application stops, and can be used for tracking memory leaks. There must be one and only one instance of this block declared in the IEC application in order to use any other *Xml* function.

2.36.2. XmlLastError

Function block Last XML error

Diagram



Outputs

iErr : DINT Error code reported by the last call:

O = OK.

other = error.

Below are possible error codes:

Error ID	Explanation
1	Invalid instance of XMLManager (should be only one)
2	Manager already open - should be only one instance declared
3	Manager not open - no instance of XmlManager declared
4	Invalid handle
5	Too many tags allocated
6	Invalid handle of a text buffer (or no instance of TxbManager declared)
7	Invalid XML file or XML syntax error
8	Cannot read file
9	Cannot write file

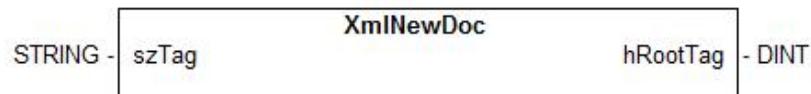
Description

All XML functions and blocks simply return a Boolean Information as a return value. This function can be called after any other function giving a *FALSE* return. It gives a detailed error code about the last detected error.



2.36.3. Reading/writing documents

Functions **XmlNewDoc** New XML document



Input szTag : STRING Name of the root tag.

Output hRootTag :
DINT Handle of the root tag. Can be used as a tag handle
for building the document, or as a complete
document handle for releasing the memory.

Description This function allocates a new XML document, initially filled with one empty root tag. The application is responsible for releasing the allocated memory by calling the **XmlFreeDoc** function.

XmlFreeDoc Free XML document

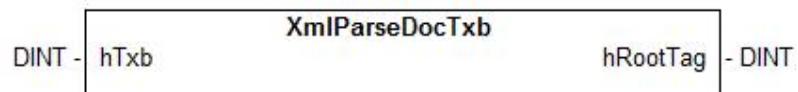


Input hRootTag :
DINT Handle of the XML document's root tag.

Output bOK : BOOL TRUE if successful.

Description This function releases the memory allocated for an XML document. It releases the specified tag and all its child tags. This function can also be used to delete a child tag and keep its parent tag in memory.

XmlParseDocTxb Parse the XML document text buffer



Input hTxb : DINT Handle of a text buffer containing XML text

Output hRootTag : DINT Handle of the root tag. Can be used as a tag handle for exploringment, or as a complete



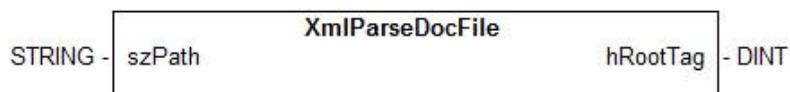
document handle for releasing the memory

Description This function parses the XML text contained in a text buffer and allocates a new XML document to store it. The application is responsible for releasing the allocated memory by calling the XmlFreeDoc function.

 **Attention**

XML parsing and writing functions support only 8 bit encoding features. This can be UTF8 or ANSI (ISO-8859-1) encoding. Some additional functions will be provided for conversion between UTF8 and ANSI formats.

XmIParseDocFile Parse the XML document file



Input szPath : STRING Full qualified pathname of the XML file.

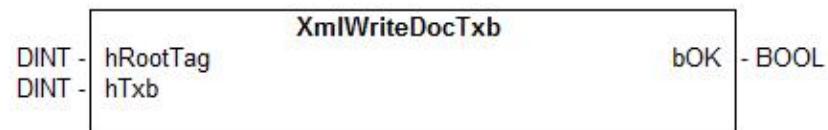
Output hRootTag : DINT Handle of the root tag. Can be used as a tag handle for exploringment, or as a complete document handle for releasing the memory.

Description This function parses the XML text contained in a file and allocates a new XML document to store it. The application is responsible for releasing the allocated memory by calling the XmlFreeDoc function.

 **Attention**

XML parsing and writing functions support only 8 bit encoding features. This can be UTF8 or ANSI (ISO-8859-1) encoding. Some additional functions will be provided for conversion between UTF8 and ANSI formats.

XmIWriteDocTxb Use XML to write to the document text buffer



Input hRootTag : DINT Handle of the XML document root tag.

hTxb : DINT Handle of the text buffer where to store XML text. This handle has been allocated and will be released by the application.



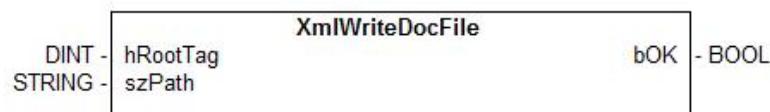
Output bOK : BOOL TRUE if successful.

Description This function format an XML document to a text buffer. It does not releases the memory allocated for the XML document.

 **Attention**

XML parsing and witing functions support only 8 bit encoding features. This can be UTF8 or ANSI (ISO-8859-1) encoding. Some additional functions will be provided for conversion between UTF8 and ANSI formats.

XmIWriteDocFile Use XML to write to the document file



Input hRootTag : DINT Handle of the XML document root tag.

szPath : STRING Full qualified pathname of the XML file to be created.

Output bOK : BOOL *TRUE* if successful.

Description This function format an XML document to a file. It does not releases the memory allocated for the XML document.

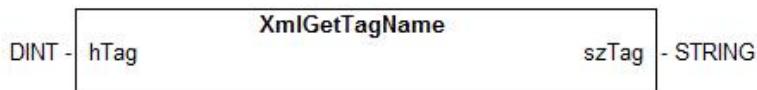
 **Attention**

XML parsing and witing functions support only 8 bit encoding features. This can be UTF8 or ANSI (ISO-8859-1) encoding. Some additional functions will be provided for conversion between UTF8 and ANSI formats.



2.36.4. Exploring an XML document

Functions **XmIGetTagName** Get the XML tag name



Input hTag : DINT Handle of the XML tag

Output szTag : STRING Name of the tag. Ex: "ABC" for <ABC/>

Description This function returns the name of an XML tag

XmICheckTagName Check the XML tag name



Input hTag : DINT Handle of the XML tag

szTag : STRING Desired tag name

Output bOK : BOOL *TRUE* if the tag matches the specified name

Description This function returns *TRUE* if the specified tag has the specified name. This function makes case insensitive comparison of tag names.

XmIGetTagCont Get the XML tag contents



Input hTag : DINT Handle of the XML tag.

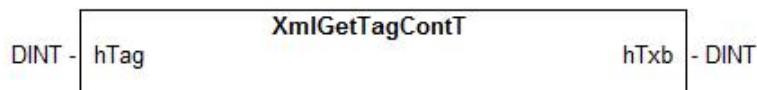
data : ANY Variable to be assigned with the contents of the tag.

Output bOK : BOOL *TRUE* if successful.

Description This function copies the contents of an XML tag to a variable. The specified variable can have any data type.



XmIGetTagContT Get the XML tag contents to a text buffer



Input **hTag** : DINT Handle of the XML tag.

Output **hTxb** : DINT Handle of the text buffer allocated and filled with tag contents.

Description This function copies the contents of an XML tag to a text buffer. The text buffer is allocated by the function, and must be released by the application after use.

XmIGetTagAttrib Get the XML tag attributes



Input **hTag** : DINT Handle of the XML tag

szAttrib : STRING Name of the desired attribute

data : ANY Variable to be assigned with the value of the tag attribute

Output **bOK** : BOOL *TRUE* if successful

Description This function copies the contents of an XML attribute to a variable. The specified variable can have any data type.



XmGetParent Get the XML parent tag

Input IN : DINT Handle of the XML tag.

Output OUT : DINT Handle of the parent tag.

Description This function returns the handle of the parent of an XML tag.

XmFirstChild XML first child



Input hTag : DINT Handle of the XML tag.

szTag : STRING Name of the desired child tag. Empty string for enumerating all children.

Output hChildTag Handle of the first child tag matching the specified name, or 0 if no child found.

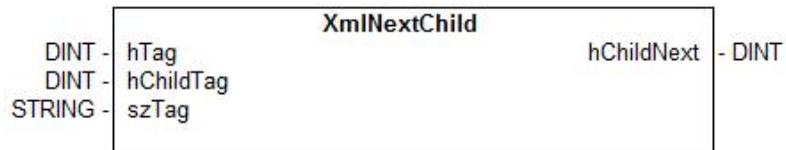
Description This function returns the handle of the first child tag having the specified name. If "szTag" is empty, then the first child tag is returned whatever its name is.

ST Usage

```
hChild := XmFirstChild (hParentTag, 'TagName');
while hChild <> 0 do
    (* explore the tag *)
    hChild := XmNextChild (hParentTag, hChild, 'TagName');
end_while;
```

XmINextChild

XML next child



Input	hTag : DINT	Handle of the XML tag
	hChildTag : DINT	Handle of the previous child tag
	szTag : STRING	Name of the desired child tag. Empty string for enumerating all children
Output	hChildNext	Handle of the next child tag matching the specified name, or 0 if no child found

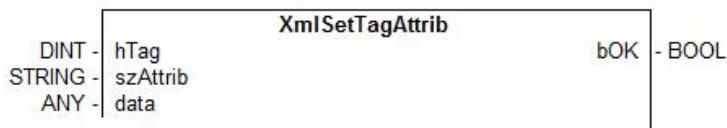
Description This function returns the handle of the next child tag having the specified name. If *szTag* is empty, then the next child tag is returned whatever its name is.

ST Useage

```
hChild := XmlFirstChild (hParentTag, 'TagName');
while hChild <> 0 do
    (* explore the tag *)
    hChild := XmINextChild (hParentTag, hChild, 'TagName');
end_while;
```

2.36.5. Building an XML document

XmISetTagAttrib Set the XML tag attributes



- Input** **hTag** : DINT Handle of the XML tag.
- szAttrib** : STRING Name of the attribute.
- data** : ANY Variable to be copied to the value of the tag attribute.
- Output** **bOK** : BOOL *TRUE* if successful.

Description This function copies the value of a variable to an XML attribute. The specified variable can have any data type.

XmISetTagCont Set the XML tag contents



- Input** **hTag** : DINT Handle of the XML tag.
- data** : ANY Variable to be copied to the contents of the tag.
- Output** **bOK** : BOOL *TRUE* if successful.

Description This function copies the value of a variable to the contents of an XML tag. The specified variable can have any data type.



XmISetTagContT Set the XML tag contents text buffer



Input **hTag** : DINT Handle of the XML tag.

hTxb : DINT Handle of the text buffer to be copied.

Output **bOK** : BOOL *TRUE* if successful.

Description This function copies a text buffer to the contents of an XML tag. The text buffer is allocated by the application, and must be released by the application after use.

XmIAddChild Add new child to XML tag



Input **hTag** : DINT Handle of the XML tag

szTag : STRING Name of the new child tag

Output **hChildTag** : DINT Handle of the new child tag.

Description This function adds a new child tag to an XML tag, with the specified tag name.

2.36.6. Exchanging variables using their name

XmISybValue Get the XML symbol value



Input **szSyb** : STRING Symbol of the variable.

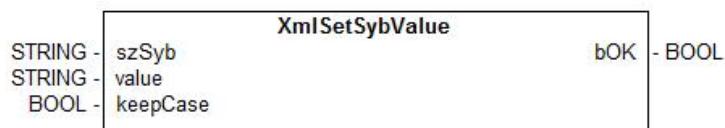
keepCase : BOOL Default is *FALSE* as normally all embedded symbols are turned uppercase by the compiler. Must be *TRUE* if the *Keep case of embedded symbols* compiling option is set.

Output **value** : STRING Current value of the variable formatted as STRING.

Description This function copies the value of the variable having the symbol specified in **szSyb** to the output string. It assumes that the symbol of the variable has been embedded in the runtime application.

This function enables the automation of data exchange between XML data and the runtime. For Example, you can find the symbol of a variable in a tag attribute and update the value in another one.

XmISybValue Set the XML symbol value



Input **szSyb** : STRING Symbol of the variable.

value : STRING Value to be assigned to the variable, formatted as STRING.

keepCase : BOOL Default is *FALSE* as normally all embedded symbols are turned uppercase by the compiler. Must be *TRUE* if the *Keep case of embedded symbols* compiling option is set.

Output **bOK** : BOOL *TRUE* if successful.

Description



2.37. T5 Registry management

The *T5 Registry* enables you to design and monitor remotely a hierarchical registry of parameters. Parameters can be set apart from the Workbench, and can also be read or written from the IEC program.

The following functions are available:

RegParGet Get the current value of a parameter

RegParPut Change the value of a parameter



2.37.1. Parameter pathnames

Any parameter is specified by a full qualified pathname that gives its exact location in the registry. The / separator is used to separate folders in the pathname. For a registry defined as:

(Root)

- Folder1
- Folder2
 - MyParam

The pathname of the parameter will be:

/Folder1/Folder2/MyParam



Attention

- T5 Registry may be not available on some targets. Please refer to OEM instructions for further details about available features.
- All parameter pathnames are case sensitive.



2.37.2. T5 Registry management functions

Functions	RegParGet	Get the current value of a parameter
Code	<code>Q := RegParGet (PATH, DEF);</code>	
	PATH : STRING	specifies the full pathname of the parameter in the registry
	DEF : ANY	default value to be returned if the parameter does not exist
	Q : ANY	current value of the parameter.
Description	Attention	
	The DEF input defines the actual type for ANY pins. If you specify a constant expression, it must be fully type-qualified according to the desired returned value. Example for getting a parameter as an INT:	
	<code>intVariable := RegParGet ('\MyParam', INT#0);</code>	
	All pathnames are case sensitive.	

Functions	RegParPut	Change the value of a parameter
Code	<code>OK := RegParPut (PATH, IN);</code>	
	PATH : STRING	specifies the full pathname of the parameter in the registry
	IN : ANY	new value for the parameter
	OK : BOOL	<i>TRUE</i> if successful
Description	The function will return <i>FALSE</i> in the following cases:	
	<ul style="list-style-type: none">The specified pathname is not found in the registry.	
	The registry is currently being saved and cannot be changed.	



Chapter 3: Embedded HMI

The Workbench enables you to design and integrate in your project an Embedded HMI application to be run on your runtime platform, assuming that it has a graphic display available. The Embedded HMI manages the animated display of screen masks designed in the Workbench with appropriate tools. Interactions between the IEC application and the embedded HMI is performed through variables (controlled by the IEC program and linked to some properties of graphic objects) plus a set of functions and function blocks.

To enable the Embedded HMI in your IEC application you should:

1. Within the Workspace when the project is selected, select and run the **Insert Shortcut / Embedded HMI** command from the contextual menu.
During this step, you must select the kind of device (hardware display) available on your system.
Refer to OEM's instructions for more information about available hardware.
2. Declare one instance of the hmiManager function block.

3.1. Embedded HMI strings

The string editor for Embedded HMI enables you to enter a collection of strings to be used as static properties among graphic objects put in screen masks. Each string is identified by a name. Using strings instead of entering texts directly in objects properties is a nice way to get the consistent set of texts used in your HMI application in one look, and eases the translation to other languages.

3.2. Embedded HMI bitmaps

In order to insert pictures (bitmaps) in screen masks, you first need to declare them in the Bitmap Editor for Embedded HMI. Each bitmap declared here is defined as:

- a name (ID) to be used in graphic objects properties and IEC programs
- a pathname to a valid .BMP file (other image formats are not supported)

If you do not specify any path for the .BMP file, it is located in the \BITMAP folder where the Workbench is installed.

Bitmap IDs can be directly used in IEC programs as integer values. For example, to set a dynamically changed bitmap to an object through a variable. You don't need to declare or define bitmap IDs in the IEC application. Their names are automatically known by the compiler.

Bitmaps are automatically converted into the set of colors available on the selected display device.

3.3. Embedded HMI character fonts

In order to use character fonts in screen masks, you first need to declare them in the Font Editor for Embedded HMI. Each font declared can be defined as:

- either a system font if available on your display device
- or a specific user-defined character font (see below)



User defined character fonts are stored in files suffixed with ".K5FNT" and created with a tool installed with the Workbench.

To create or change a specific character font run the "Font Editor" command from the popu menu.

If you do not specify any path for a font file, it is located in the \FONT folder where the Workbench is installed.

Font IDs can be directly used in IEC programs as integer values. For example, to set a dynamically changed font to an object through a variable. You do not need to declare or define font IDs in the IEC application. Their name are automatically known by the compiler.

3.4. Embedded HMI screens

The Embedded HMI application is designed as a set of screen masks. To create a new screen, run the "Insert Item / New Screen" command from the contextual menu in the Workspace. Each screen is identified by a name (ID) such as shown in the workspace tree. You can change this ID using the "Rename" command.

The hmiSetScreen function enables you to dynamically change the active screen mask from IEC programs.

Screen IDs can be directly used in IEC programs as integer values. You do not need to declare or define again IDs in the IEC application. Their name are automatically known by the compiler.

Double click on a screen in the Workspace to edit its contents. A screen mask is designed as:

- some properties visible when no object is selected,
- a set of graphic objects having static or dynamic properties.



3.5. Embedded HMI graphic objects

A graphic object is a predefined piece of graphic to be inserted into a screen mask. Each object has a set of properties that can be either statically defined or dynamically set by the IEC program through the value of a variable. The following kinds of objects are available:

- Rectangle (shape or text area)
- Bargraph
- Picture (bitmap)

Available objects are displayed in the bottom-right area of the screen editor. Just drag them to the screen and then fill up their properties.

3.5.1. Rectangle (or text area)

This object is used for the display of a rectangle area that may contain a single line of text. Its properties are:

Property	Meaning
Align	Horizontal alignment for the text (left, center or right). If dynamically passed through a variable, use these values: 0 = center -1 = left 1 = right
Border color	Color index of the border (can be dynamically passed through a variable)
Border size	Width of the border in pixels (0 = no border)
Back. color	Background color index (can be dynamically passed through a variable)
Text color	Color of the text (can be dynamically changed through a variable)
Text	Text to be displayed: can be a specific text, or an ID of a declared HMI string, or the name of a STRING variable
Font	Character font: can be an ID of a declared font or the name of an integer variable that will contain this ID.



3.5.2. Round rectangle

This object is used for the display of a rounded rectangle area that may contain a single line of text. Its properties are:

Property	Meaning
Align	Horizontal alignment for the text (left, center or right). If dynamically passed through a variable, use these values: 0 = center -1 = left 1 = right
Border color	Color index of the border (can be dynamically passed through a variable)
Border size	Width of the border in pixels (0 = no border)
Back. color	Background color index (can be dynamically passed through a variable)
Text color	Color of the text (can be dynamically changed through a variable)
Text	Text to be displayed: can be a specific text, or an ID of a declared HMI string, or the name of a STRING variable
Font	Character font: can be an ID of a declared font or the name of an integer variable that will contain this ID.
Round corner	Size of the round corner, in pixels

3.5.3. Ellipse

This object is used for the display of an ellipse or circle. Its properties are:

Property	Meaning
Border color	Color index of the border (can be dynamically passed through a variable)
Border size	Width of the border in pixels (0 = no border)
Fill color	Background color index (can be dynamically passed through a variable)
Font	Character font: can be an ID of a declared font or the name of an integer variable that will contain this ID.
Text color	Color of the text (can be dynamically changed through a variable)
Text	List of choices (lines) to be displayed, separated by ' ' characters. E.g.: Choice1 Choice2 Choice3



3.5.4. Line

This object is used for the display of a line. Its properties are:

Property	Meaning
Direction	Direction of the line within its rectangle area
Line width	Width of the line, in pixels
Status	Boolean status for color selection
Color FALSE	Line color when status is FALSE
Color TRUE	Line color when status is TRUE

3.5.5. Bargraph

This object is used for the display of a bargraph reactangle area. Its properties are:

Property	Meaning
Direction	Growing direction of the bargraph. If dynamically passed through a variable, use these values: 0 = left to right 1 = bottom to top 2 = right to left 3 = top to bottom
Border color	Color index of the border (can be dynamically passed through a variable)
Border size	Width of the border in pixels (0 = no border)
Back. color	Background color index (can be dynamically passed through a variable)
Fill color	Color of the filled part (can be dynamically changed through a variable)
Fill value	Percentage of the rectangle filled (from 0 to 100), generally exchanged through an integer variable.

3.5.6. Picture (bitmap)

This object is used for the display of a bitmap picture. Its properties are:

Property	Meaning
Back. color	Background color index used if the object is larger than the bitmap.
Bitmap	ID of the bitmap declared in the bitmap editor. Can be the name of an integer variable that will contain this ID.



3.5.7. Trend

This object is used for the display of a trend diagram. Its properties are:

Property	Meaning
Style	Kind of drawing. Possible values are: 1: histogram (filled area under the curve) 2: points (curve line)
Border color	Color index of the border (can be dynamically passed through a variable)
Border size	Width of the border in pixels (0 = no border)
Back. color	Background color index (can be dynamically passed through a variable)
Fill color	Color of the filled part (can be dynamically changed through a variable)
Item width	Number of pixels on the X axis for each sample of the trend
First array item	0-based index of the item of the array to be displayed as the first value
Last array item	0-based index of the item of the array to be displayed as the last value
Array	Array of values to be displayed (in range 0-100)

3.5.8. Menu

This object is used for the display of menu like scrollable choice list. Its properties are:

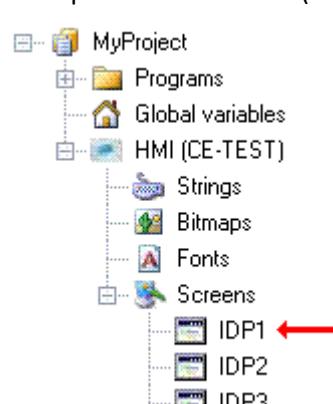
Property	Meaning
Align	Horizontal alignment for the text (left, center or right). If dynamically passed through a variable, use these values: 0 = center -1 = left 1 = right
Border color	Color index of the border (can be dynamically passed through a variable)
Border size	Width of the border in pixels (0 = no border)
Font	Character font: can be an ID of a declared font or the name of an integer variable that will contain this ID.
Back. color	Background color index (can be dynamically passed through a variable)
Text color	Color of the text (can be dynamically changed through a variable)
Select font	Character font for the selected line
Select back. color	Background color index of the selected line



Property	Meaning
Select text color	Color of the text on the selected line
Text	List of choices (lines) to be displayed, separated by ' ' characters. E.g.: Choice1 Choice2 Choice3
Line height	Height of a line, in pixels
Selection	Index of the selected line (first line is 1) Any selection is removed if this value is 0

3.6. Embedded HMI Functions and Function blocks

A set of functions and function blocks is available for the dynamic control of the Embedded HMI application from the IEC programs.

Function Block	hmiManager	Enable the use of Embedded HMI
Outputs	OK : BOOL	<i>TRUE</i> if Embedded HMI is active
Description	You need to declare one and only one instance of this function block in order to activate the Embedded HMI. It is not mandatory to call the instance from your program. Declaring the instance is enough for activation.	
hmiSetScreen	Set the active screen for the Embedded HMI	
Inputs	ID : DINT	Identifier of the screen
	DISPLAY : DINT	This parameter is reserved for extensions and must be 0
Outputs	OK : BOOL	<i>TRUE</i> if successful
Description	Call this function in order to set the active screen of the Embedded HMI. The screen ID is the name given to the screen such as displayed in the project tree. Example - hmiSetScreen (IDP1, 0);	
		



3.7. Managing menus

The system includes a set of functions function blocks for easily managing hierarchical menus within an embedded HMI application. These functions are intended to be used together with the "Menu" graphic object for embedded HMI screens.



Attention

Menu management functions use safe dynamic memory allocation that needs to be configured in the project settings. From the project settings, press the "Advanced" push button and go to "Memory" tab. Here you can setup the memory for safe dynamic allocation.

3.7.1. Creating a hierarchy of menus

The following functions are used to create menus. They should be called once when the application starts so that the hierarchy of menus remain available in memory for use with the "Menu" graphic object. The following functions are available:

```
hMenu := HmiNewMenu (hParentMenu, strTitle); (* creates a main or sub menu *)
hItem := HmiNewMenuItem (hParentMenu, strChoice, idCommand); (* adds a choice to a menu *)
ok := HmiRenameMenu (hItem, strChoice); (* changes the text of a choice or the title of a menu *)
ok := HmiDelMenu (hItem); (* deletes a menu item of a menu *)
```

3.7.1.1. Arguments

```
hMenu : BOOL; (* handle of a main or sub menu *)
hParentMenu : DINT; (* handle of the parent menu - specify 0 to create a root menu *)
hItem : DINT; (* handle of a menu or an item to be used in other function calls *)
strTitle : STRING; (* optional title for a menu *)
strChoice : STRING; (* choice text for a menu item *)
idCommand : DINT; (* numerical identifier of a menu item to be used in the application *)
ok : BOOL; (* TRUE if successful *)
```

3.7.2. Running a hierarchy of menus

Then you can use the HmiMenu function block to activate the hierarchy of menus. First you must declare an instance of it, for Example:

```
MENU : HmiMenu;
```

You should call the FB instance with the following input arguments:

```
UP : BOOL; (* the selection is moved up on a rising edge of this signal *)
DN : BOOL; (* the selection is moved down on a rising edge of this signal *)
OK : BOOL; (* the selected sub-menu is selected on a rising edge of this signal *)
ESC : BOOL; (* the parent menu is selected on a rising edge of this signal *)
Menu : DINT (* handle of the root menu *)
```



Below are the outputs of the HmiMenu function block:

SelID : DINT; (* ID of the selected menu item on a rising edge of the OK input *)
SelPos : DINT; (* position of the selected choice to be passed to the "Menu" graphic object *)
Title : STRING; (* title of the current menu *)
Texts : STRING; (* list of choices to be passed to the "Menu" graphic object *)

3.7.2.1. Example

Below is an Example that creates a main menu with 2 sub-menus:

```
if not bInitDone then
    // create the main menu (root)

    hRootMenu := hmiNewMenu (0, 'Main menu');
    // create the first sub menu with 2 items

    hSubMenu := hmiNewMenu (hRootMenu, 'Menu 1');
    hmiNewMenuItem (hSubMenu, 'Choice 1', 101);
    hmiNewMenuItem (hSubMenu, 'Choice 2', 102);
    // create the second sub menu with 2 items

    hSubMenu := hmiNewMenu (hRootMenu, 'Menu 2');
    hmiNewMenuItem (hSubMenu, 'Choice 3', 201);
    hmiNewMenuItem (hSubMenu, 'Choice 4', 202);

    bInitDone := true;

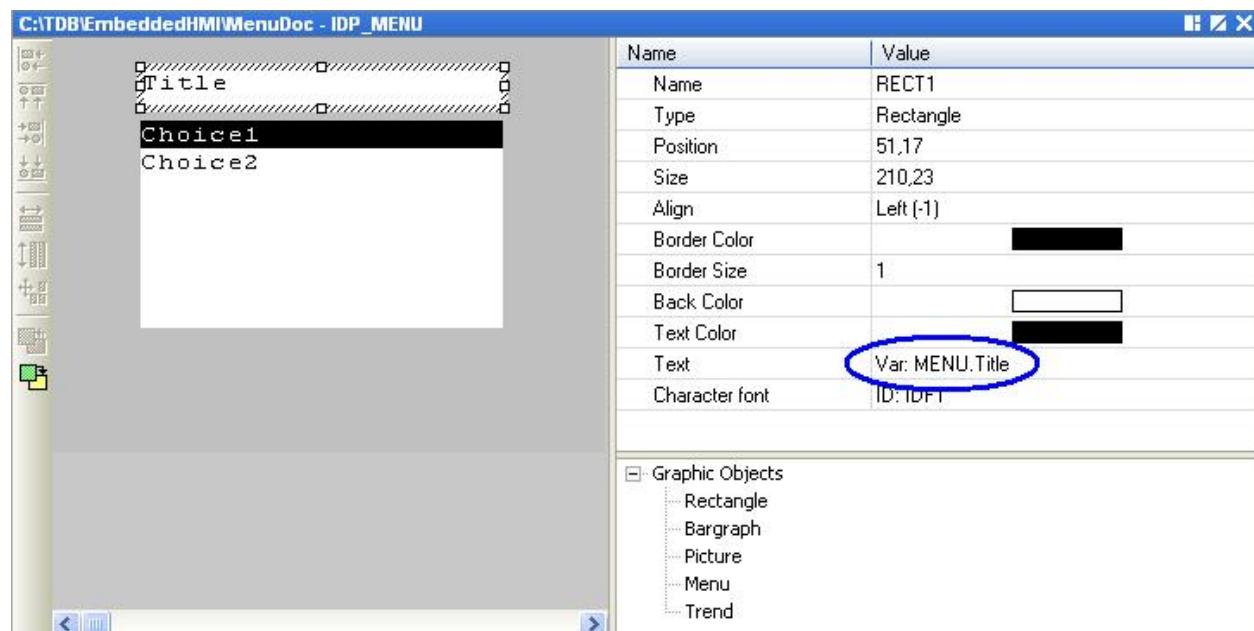
end_if;
```

Below is a typical call to the HmiMenu function block ("MENU" is a declared instance of the FB):

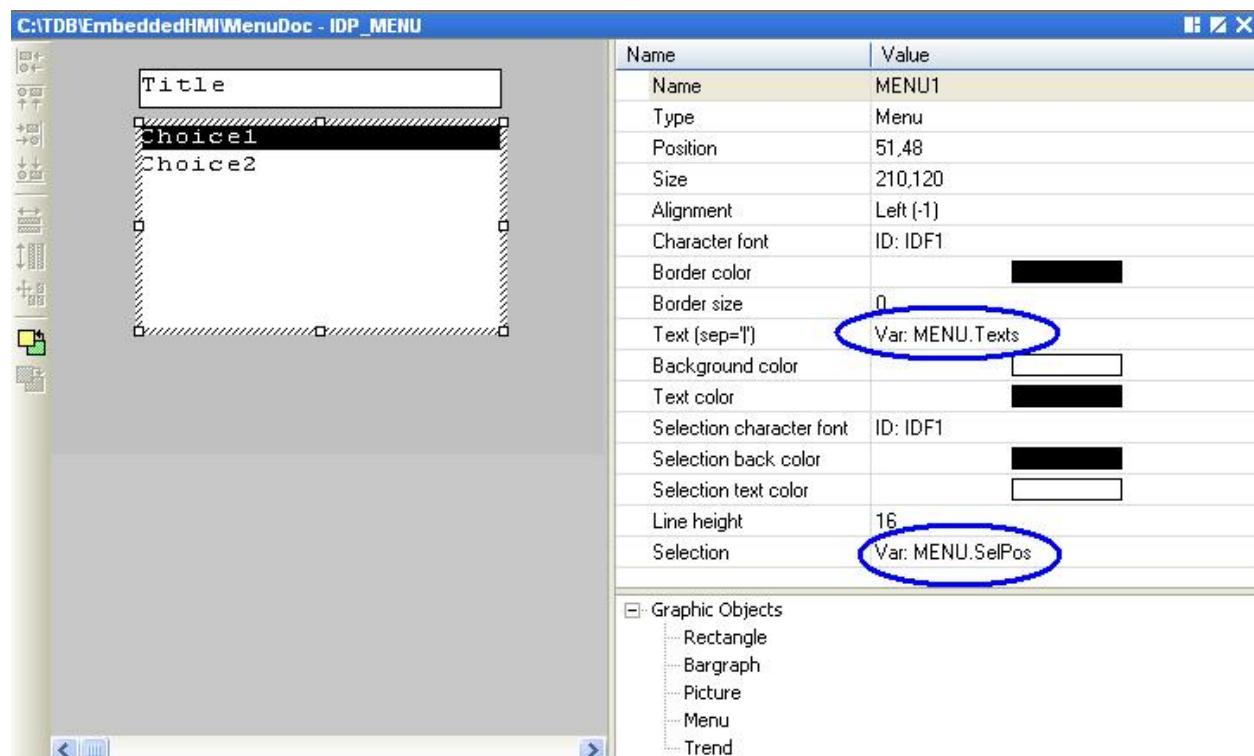
```
MENU (bUP, bDOWN, bOK, bBACK, hRootMenu);
case MENU.SelID of
    101:// manage command 1
    102:// manage command 2
    201:// manage command 3
    202:// manage command 4
end_case;
```



Below is how to configure a text graphic object to display the current menu title using the outputs of the function block instance:



Below is how to configure a "Menu" graphic object to display the current menu title using the outputs of the FB instance:





Chapter 4: T5 Registry for runtime parameters

The *T5 Registry* enables you to design and monitor remotely a hierarchical registry of parameters. Parameters can be set apart from the Workbench, and can also be read or written from the IEC program.

4.1. Designing the registry of parameters

The Registry Design tool enables you to design the set of runtime parameters and how they will be edited during monitoring. The definition of parameters is stored in an XML file. The design tool works mainly on this file. Additionally, the design tool is used to send the new registry to the runtime in binary form. Optionally the XML file can also be stored in the runtime.

To run the design tool from the Workbench, use the menu command Tools / Runtime Parameters / Design.

Parameters are freely organized with folders. The left part of the editor shows you the complete hierarchy of folders and parameters. The right hand area is used for entering the detailed description of the folder or parameter currently selected in the tree. Use the commands of the **Edit** menu to add new folders and parameters.

For each folder you must specify the following pieces of Information:

Information	Description
Name	Name of the folder - cannot contain "/" characters.
Access	Access mode (read-write or read only).
Protection	Protection mode for write (using password): <i>No</i> = no protection. <i>Yes</i> = password protected. <i>Inherit</i> = use the same protection as the parent folder.
Password	Password for write.
Description	Free description text.

For each parameter you must specify the following pieces of Information:

Information	Description
Name	Name of the folder - cannot contain / characters.
Type	Data type.
Max. length	Maximum length for STRING. Maximum length cannot exceed 200 characters.
Access	Access mode (read-write or read only).
Protection	Protection mode for write (using password): <i>No</i> = no protection.



Information	Description
	<i>Yes</i> = password protected. <i>Inherit</i> = use the same protection as the parent folder.
Password	Password for write.
Default	Default value when registry is loaded for the first time.
Description	Free description text.
Editing mode	Editing method when monitoring.
Extra data	Description of choices for a list or combo box editing mode.
Minimum	Minimum allowed value for numbers.
Maximum	Maximum allowed value for numbers.

The commands of the **Project** menu are used for updating the runtime:

- Check Registry

This command checks the whole contents of the designed registry and reports possible consistency errors.

- Send Registry

This command sends the registry in binary format to the runtime system. You normally need to select the communication parameters of a remote runtime, but you can also save the binary registry in a local file if you want to use another transfer method. For remote sending, you can optionally send the XML definition file together with the binary registry.

Depending on the runtime system, it may happen that the new registry is not taken into account immediately, if it is currently in use by the system. Some runtimes will need a Stop/Restart of the IEC application. Some other runtimes may require a full reboot. Refer to the OEM instructions.

4.2. Monitoring parameters

The *Register Host* tool is used for monitoring runtime parameters On Line. Parameters can be displayed, and possibly modified according to their protection as defined in the Design tool. To run the host tool from the Workbench, use the menu command Tools / Runtime Parameters / Monitor. Then use the File / Open command to connect to the runtime and monitor its parameters.

The left side tree shows the folders of the Registry. The right-side area shows the parameters of the selected folder. Double click on a parameter to change its value.

Use the View / Refresh command to refresh the value of the parameters.

The File / Save menu command asks the runtime system to save the contents of the registry to backup support (flash or disk).

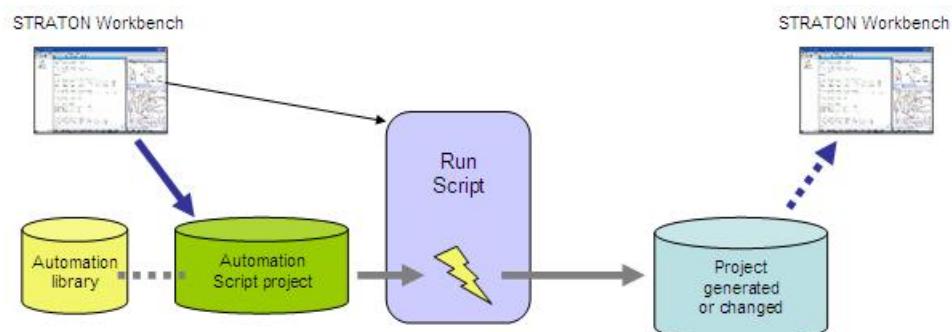
Chapter 5: Project Automation Library

The workbench enables you to automate the creation or change of your IEC61131-3 projects. Many powerful applications are possible by writing automation scripts using ST language:

- Create new wizards to build the skeleton of a new project
- Create wizards that automate or import the I/O configuration
- Automatically generate documentation about project items
- Any import / export procedure

No extra tools are required, or specialist knowledge of other programming or scripting languages such as VB or C++. Simply use the Workbench to develop, test and run your scripts. Scripts are written using IEC languages (typically Structured Text).

A project automation script is a program written in IEC languages. Structured Text is typically used as the programming language for scripts as it is the most adapted to automation feature. The script is developed as a project, linked to a dedicated library called *AUTOMATION*:



When the Workbench is used for the development of a project automation script, the *Simulation* and *On Line* commands are replaced by a *Execute Script* command used for executing the script.



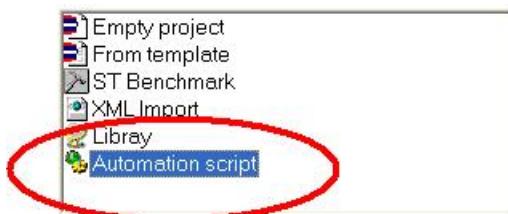
Unlike other IEC projects, the program of a script is executed only once, and is not repeated from cycle to cycle.

Scripts can be used either for generating the skeleton of a new project, or for changing / completing an existing project. Thus the AUTOMATION library does not only contain functions for building the project, but also for enumerating and changing the existing items of a project.



5.1. Creating a new Project Automation script

To create a new Project Automation script, Run the File / New Project command from the menu, and then select the **Automation Script** choice in the project creation dialog box:



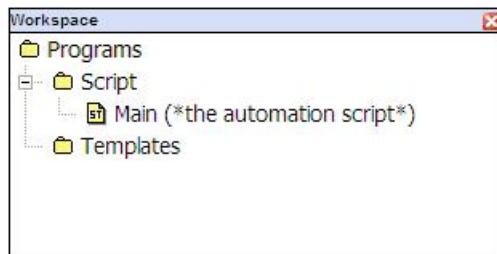
Then follow the instructions. You must specify how your script will be used:

- Generating a new project from scratch.
- Modifying an existing project.
- Modifying a project or creating it if not yet existing.

While creating the script, you already can define some parameters to be entered by the user when the script is run. The system will automatically create:

- Some global variables in your script project that will be the parameters.
- A list of variables grouping parameters.
- The few lines of ST code for prompting the user to enter parameters.

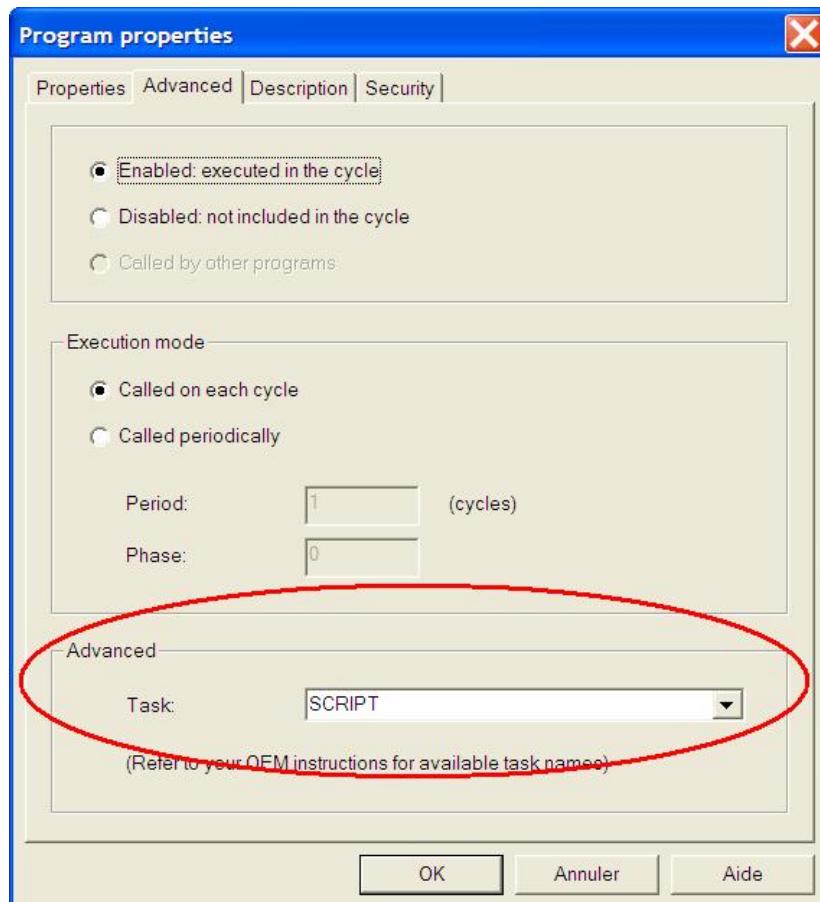
The script is generated with the following workspace:



The **Main** program is the automation script. You can freely create sub-programs and UDFBs to be called by this program.

The Templates folder will contain all your template programs to be instantiated (copied) to the target project when the script is run.

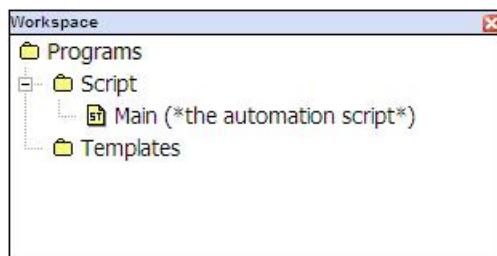
It is recommended to keep only one top level program for the script (the **Main** program). This program must have the following *Task* specification entered in its properties:



Thus, if you need to create other top level programs to be executed in your script, you need to set this property for all of them.

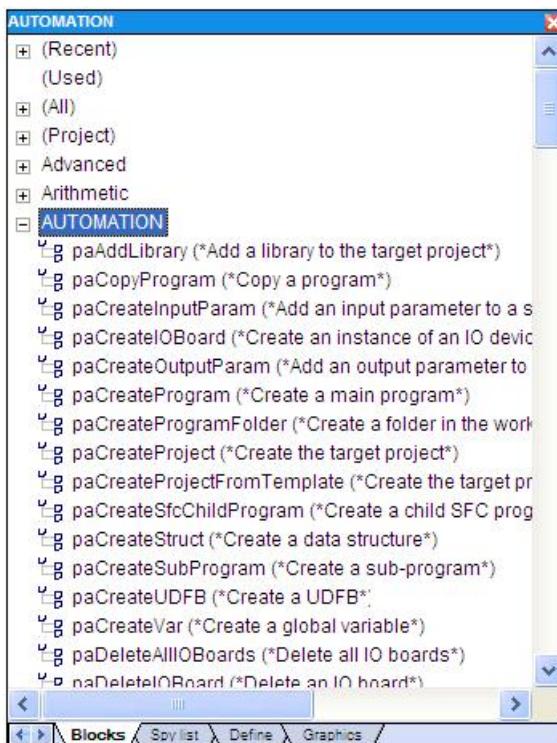
5.2. Developing and testing the script

An *Automation Script* is developed as any project with the Workbench, but with some particular constraints. The biggest difference is that a script is a program (generally written in ST language) that is executed only once, and not repeated from cycle to cycle. The top program must have the special task property set to *SCRIPT*. However, you don't need to care about that as the **New Script** wizard has generated the skeleton of the script project for you:



The **Main** program is the automation script. You can freely create sub-programs and UDFBs to be called by this program, but it is recommended to keep only one top level program for the script (the **Main** program).

The script project refers to a special library called *AUTOMATION* that contains all necessary functions for developing scripts:

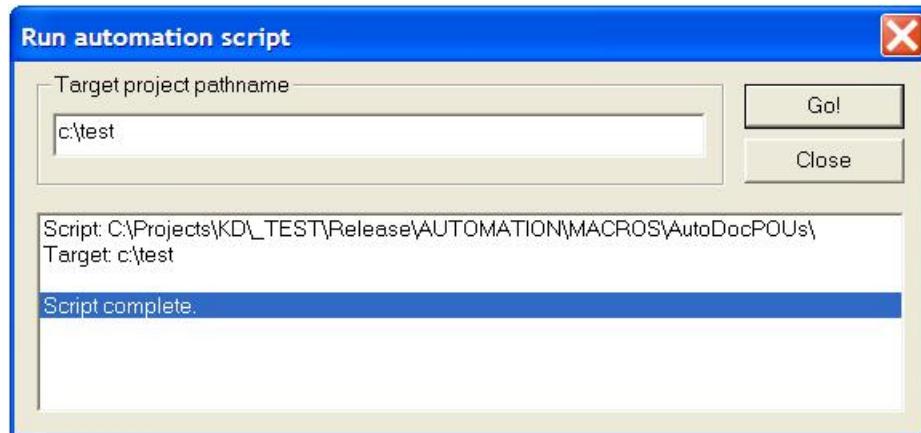


When developing a script project, the commands *Simulate* and *On Line* of the **Project** menu are replaced by the *Execute Script* command:





This command enables to run your script for test purpose. It opens the following box:

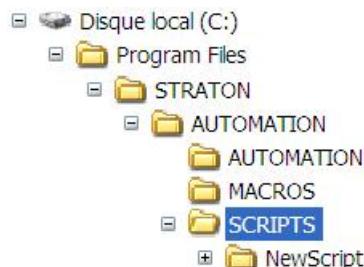


Enter the name of the target project (to be created or changed) in the upper box and press **Go!**. Any trace message or error report is displayed in the list below.

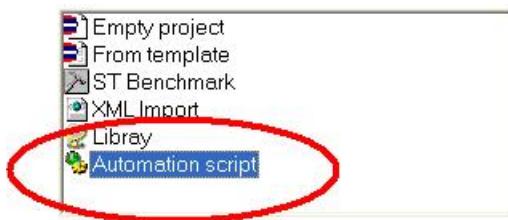
5.2.1. Registering the script in the workbench as a "New project" wizard

If your script is intended to create the skeleton for a new project, you can register it so that the workbench proposes it in its standard *New Project* procedure.

Simply copy the script folder to the AUTOMATION\SCRIPTS\ folder where the Workbench is installed:



Then your script will be available for creating a new project from the workbench. For that the user will have to select the *Automation script* choice in the **New Project** dialog box:



Your script must be compiled in order to be available as a wizard.

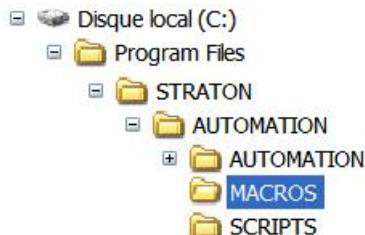
It is highly recommended that you copy the whole contents of your script project so that all possibly required template programs are available. If your script does not use templates, only the file APPLI.XTI is required in the script folder.



5.2.2. Registering the script in the workbench as a tool

If your script is intended to modify an open project, you can register it so that the workbench proposes it from the Tools / Automation Script menu command.

Simply copy the script folder to the AUTOMATION\MACROS\ folder where the Workbench is installed:



Then your script will be available for modifying a project open with the workbench.

It is highly recommended that you copy the whole contents of your script project so that all possibly required template programs are available. If your script does not use templates, only the file APPLI.XTI is required in the script folder.

5.2.3. Running a script from other applications

Scripts developed with the workbench can be run from external applications. Below are possible calling methods:

Explanation	Arguments
From the command line	
Synt.	K5Script.exe <script> <target_project>
	Scripts can be executed directly from the Windows console or from .BAT files, using the K5Script.exe utility program installed with the Workbench. <script> : Name of the folder containing the script. If no pathname is specified, the script folder is searched under the AUTOMATION\SCRIPTS\ folder where the Workbench is installed. <target_project> : full qualified pathname of the target project to be created or modified by the script.
From application using COM interface	
Synt.	Not available yet.
From "C / C++" applications	
Synt.	From "C" language: LPCSTR __declspec(dllexport) K5NETAS_RunScript (LPCSTR szScript, LPCSTR szDest); From "C++" language: extern "C" LPCSTR __declspec(dllexport) K5NETAS_RunScript (LPCSTR szScript, LPCSTR szDest);



Explanation

Your "C/C++" applications can link to the K5NETAS.DLL located with the installed Workbench and use the following export function to run a script.

Arguments

szScript : Name of the folder containing the script. If no pathname is specified, the script folder is searched under the "AUTOMATION\SCRIPTS\" folder where the Workbench is installed.

szDest : full qualified pathname of the target project to be created or modified by the script

Return value:

The function returns a pointer to a string containing the report.



5.3. Reference

Below are the functions available in the *AUTOMATION* library for developing automation scripts:

Main services	Target project management
Declaring and managing IEC objects	<ul style="list-style-type: none">• Declaring programs• Declaring data types (structures)• Declaring variables• Declaring I/O boards
Generating project documents (files)	Common file handling services Text files IEC 61131-3 source files Network and fieldbus configuration Watch window documents Resources
Generating documents from templates	Defining variable keywords for a template Templates of programs
Tools	Working with script parameters Miscellaneous
Project level services	Below are the main services to manage the target project
	Opening the target project
paCreateProject	Create the target project
paCreateProjectFromTemplate	Create the target project using a template
paOpenProject	Open the target project for modification
	Setting project level Information
paSetProjectComment	Set the description for title bar
paSetOption	Set a project option
paAddLibrary	Add a link to a library



5.3.1. Project Level Services

Functions	paCreateProject	Create the target project
	Syntax	<code>OK := paCreateProject ()</code>
	Parameter	<code>OK : BOOL;</code> TRUE if successful.
	Description	This function creates the target project as an empty project. The destination folder should not exist yet. This function is typically called in a script that creates the skeleton of a new project. You should call this function at the beginning of your script before any other declaration or creation function. Your script should terminate if this function returns <i>FALSE</i> .
	Example	<pre>// at the beginning of the script: if not paCreateProject () then return; end_if; // ...</pre>
Functions	paCreateProjectFromTemplate	Create the target project from a template
	Syntax	<code>OK := paCreateProjectFromTemplate (TEMPLATE)</code>
	Parameter	<code>TEMPLATE : STRING;</code> Name of a template projects
	Parameter	<code>OK : BOOL;</code> TRUE if successful
	Description	This function creates the target project as a copy of a template project. The destination folder should not exist yet. This function is typically called in a script that creates the skeleton of a new project. Template projects are located under the <code>TEMPLATE\</code> folder of the installed Workbench. the <i>TEMPLATE</i> parameter specifies only the name of the template project, and does not include its path. You should call this function at the beginning of your script before any other declaration or creation function. Your script should terminate if this function returns <i>FALSE</i> .
	Example	<pre>// at the beginning of the script: if not paCreateProjectFromTemplate ('MyTemplate') then return; end_if;</pre>



// ... // ...



Functions	paOpenProject	Open the target project for modification
	Syntax	OK := paOpenProject (CREATE)
	Parameter	CREATE : BOOL; If <i>TRUE</i> , creates the project if not existing yet
		OK : BOOL; <i>TRUE</i> if successful
	Description	This function opens the target project as an empty project. If the CREATE parameter is <i>FALSE</i> , the project must exist. If CREATE is <i>TRUE</i> and the project does not exist yet, then it is created. You should call this function at the beginning of your script before any other declaration or creation function. Your script should terminate if this function returns <i>FALSE</i> .
	Example	<pre>// ensure that the target project already exists if not paOpenProject (FALSE) then return; end_if; // ...</pre>
Functions	paSetProjectComment	Change the description of the target project
	Syntax	OK := paSetProjectComment (COMM)
	Parameter	COMM : STRING; Project description
		OK : BOOL; <i>TRUE</i> if successful
	Description	This function sets the comment text used for describing the project. This text is displayed in the title bar of the Workbench when the project is open. Note that in case of a new project wizard, a comment text has already been entered by the user.
	Example	<pre>// after opening or creating the target project paSetProjectComment ('Project generated by script')</pre>



Functions **paSetOption** Set a project option**Syntax** **OK := paSetOption (OPTION, VALUE)****Parameter** **OPTION : STRING;** Option name**VALUE : STRING;** Value of the option**OK : BOOL;** *TRUE* if successful**Description** This function sets an option in the project settings. The following options are available:

Name	Value	Description
TARGET	T5RTM / T5RTI	Type of target runtime
DEBUG	ON / OFF	Compile in Debug mode
CTSEG	ON / OFF	Complex variables in a separate segments
FBDFLOW	ON / OFF	Color FBD lines during debug
EMBEDSYMBOLS	ON / OFF	Embed all variable symbols
EMBEDSYBCASE	ON / OFF	Keep case of embedded symbols
WARNING	ON / OFF	Display warning messages
SFCSAFE	ON / OFF	Check SFC charts safety
SPCLEAN	ON / OFF	Remove code of uncalled sub-programs
IECCHECK	ON / OFF	Check conformity to IEC standard
SAFEARRAY	ON / OFF	Check array bounds at runtime
LARGEJUMP	ON / OFF	Allow large jump instructions
LOCK	IO / ALL / NONE	Range of lockable variables
PASSWORD	number	Password for connection
Name	Value	Description

Example



Functions paAddLibrary

Add a link to a library

Syntax `OK := paAddLibrary (PATH)`

Parameter `PATH : STRING;` Full qualified pathname of the library folder

`OK : BOOL;` *TRUE* if successful

Description This function adds a link to a library for the target project.

Example `// after opening or creating the target project
paSetProjectComment ('Project generated by script')`



5.4. Declaring Programs

Below are the main services to declaring programs.

Declaration	paCreateProgram - Create a main program paCreateSubProgram - Create a sub-program paCreateUDFB - Create User Defined Function Block paCreateSfcChildProgram - Create a child SFC program paCopyProgram - Duplicate a program paSetProgramComment - Set the description text of a program
Working with existing programs	paEnumProgram - Enumerate programs paGetProgramDesc - Get Information about a program paDeleteProgram - Remove a program
Arranging programs in folders	paCreateProgramFolder - Create a new folder paSendProgramToFolder - Put a program in a folder paEnumProgramFolder - Enumerate program folders paDeleteProgramFolder - Remove a folder



5.4.1. paCreateProgram

Function Create a main program

Syntax `OK := paCreateProgram (NAME, LANGUAGE)`

Parameter	Description
NAME : STRING;	Name of the new program
LANGUAGE : STRING;	Programming language (see notes)
OK : BOOL;	TRUE if successful

Description This function creates a main program in the target project. Main programs are arranged in the cycle in the same order you create them from your script.

The following values are predefined for the *LANGUAGE* parameter:

Value	Description
_LG_SFC	Sequential Function Chart
_LG_FBD	Function Block Diagram
_LG_LD	Ladder Diagram
_LG_ST	Structured Text
_LG_IL	Instruction List

Example

```
// create a new program folder
paCreateProgramFolder ('Folder1') then
    // create a new program in LD language
    if paCreateProgram ('Prog1', _LG_LD) then
        // and put it in the folder
        paSendProgramToFolder ('Prog1', 'Folder1');
    end_if;
```



5.4.2. paCreateSubProgram

Function Create a sub-program

Syntax `OK := paCreateSubProgram (NAME, LANGUAGE)`

Parameter	Description
NAME : STRING;	Name of the new program
LANGUAGE : STRING;	Programming language (see notes)
OK : BOOL;	TRUE if successful

Description This function creates a sub-program in the target project.

The following values are predefined for the *LANGUAGE* parameter:

Value	Description
_LG_FBD	Function Block Diagram
_LG_LD	Ladder Diagram
_LG_ST	Structured Text
_LG_IL	Instruction List

Example

```
// create a new program folder
paCreateProgramFolder ('SubPrograms') then
    // create a new sub-program in LD language
    if paCreateSubProgram ('SP1', _LG_LD) then
        // and put it in the folder
        paSendProgramToFolder ('SP1', 'SubPrograms');
    end_if;
```



5.4.3. paCreateUDFB

Function Create a User Defined Function Block

Syntax **OK := paCreateUDFB (NAME, LANGUAGE)**

Parameter	Description
NAME : STRING;	Name of the new program
LANGUAGE : STRING;	Programming language (see notes)
OK : BOOL;	TRUE if successful

Description This function creates a User Defined Function Block in the target project.

The following values are predefined for the *LANGUAGE* parameter:

Value	Description
_LG_FBD	Function Block Diagram
_LG_LD	Ladder Diagram
_LG_ST	Structured Text
_LG_IL	Instruction List

Example

```
// create a new program folder
paCreateProgramFolder ('UDFBs') then
    // create a new sub-UDFB in LD language
    if paCreateUDFB ('FB1', _LG_LD) then
        // and put it in the folder
        paSendProgramToFolder ('F1', 'UDFBs');
    end_if;
```



5.4.4. paCreateSfcChildProgram

Function Create a child SFC program

Syntax `OK := paCopyProgram (SRC, DST)`

Parameter	Description
NAME : STRING;	Name of the new child program
PARENT : STRING;	Name of the parent program
OK : BOOL;	TRUE if successful

Description This function creates a new SFC program to be a child of the specified parent SFC program. The parent SFC program must exist before calling this function.:

Example

```
// create the parent program
paCreateProgram ('MAIN', _LG_SFC);

// create the child program
paCreateSfcChildProgram ('Child1', 'MAIN');
```

5.4.5. paCopyProgram

Function Duplicate a program

Syntax `OK := paCreateUDFB (NAME, PARENT)`

Parameter	Description
SRC : STRING;	Name of the source program.
DST : STRING;	Destination program.
OK : BOOL;	TRUE if successful.

Description This function duplicates the *SRC* program in the *DST* program. This function creates the *DST* program and is normally not used for overwriting an existing program.



5.4.6. paSetProgramComment

Function Set the description a program

Syntax `OK := paSetProgramComment (NAME, COMM)`

Parameter	Description
NAME : STRING;	Name of the new child program.
PARENT : STRING;	Name of the parent program.
OK : BOOL;	TRUE if successful.

Description This function sets the comment text of a program. It may be used for either programs or sub-programs or User Defined Function Blocks (UDFBs).

Example

```
// create the parent program
paCreateProgram ('MAIN', _LG_SFC);

// create the child program
paCreateSfcChildProgram ('Child1', 'MAIN');
```



5.4.7. paEnumProgram

Function Enumerate programs of the target project

Syntax **Inst_paEnumProgram (LOAD, CHECK, ITEM, FILTER)**

	Parameter	Description
	LOAD : TRUE;	If TRUE, load the list of programs.
	CHECK : BOOL;	If TRUE, open a checklist box to select some of the loaded programs.
	ITEM : DINT;	Index of the item wanted on input (1 based).
	FILTER : STRING;	Filtering mask for loading programs.
Output	NB : DINT;	Number of programs
	Q : STRING;	Name of the item selected with ITEM
Description	This function block is used for enumerating the programs, sub-programs and UDFBs of the target project. You must first call it with LOAD input at <i>TRUE</i> in order to load the list of programs. You get the number of programs in the NB output. Then call it again with LOAD at <i>FALSE</i> and specifying the index of the desired program in the ITEM input to get the name of the selected item in the Q output. The numbering of items starts at 1. When loading programs (LOAD inputs is <i>TRUE</i>), you can specify a filtering string that may include some '*' or '?' wildchars. After loading, if can call again the block with the CHECK input at <i>TRUE</i> for opening a dialog box where the user can check desired programs. After the box is closed, only checked items remain in the loaded list.	

Example

```
// "ENU" is a declared instance of paEnumProgram function block
// load all programs
ENU (TRUE, FALSE, 0, "*");

// open a dialog box for the user to check desired items
ENU (FALSE, TRUE, FALSE, 0, "");

// enumerate checked items
for i := 1 to ENU.NB do
    // select the item "i"
    ENU (FALSE, FALSE, i, "");

    // get the name of the item specified by "i"
    sProgName := ENU.Q;

end_for;
```



5.4.8. paGetProgramDesc

Function Get Information about a program

Syntax **Inst_paGetProgramDesc (NAME)**

NAME : STRING; Name of the new program
LANGUAGE : STRING; Programming language (see notes)
OK : BOOL; TRUE if successful

Output OK : BOOL; TRUE if successful

KIND : STRING; Kind of POU
LANGUAGE : STRING; Programming language
PARENT : STRING; Parent SFC program or empty string
COMMENT : STRING; Description text

Description This function block is for getting Information about a program of the target project.
It can be used for programs, sub-programs or UDFBs.

The following values are predefined for the *LANGUAGE* output:

Value	Description
_LG_SFC	Sequential Function Chart
_LG_FBD	Function Block Diagram
_LG_LD	Ladder Diagram
_LG_ST	Structured Text
_LG_IL	Instruction List

The following values are predefined for the *KIND* output:

_POU_MAIN Main program
_POU_SP Sub-program
_POU_UDFB User Defined Function Block
_POU_CHILDOF Child SFC program (the name of its parent program is in the PARENT output)

Example

```
// create a new program folder
paCreateProgramFolder ('SubPrograms') then
    // create a new sub-program in LD language
    if paCreateSubProgram ('SP1', _LG_LD) then
        // and put it in the folder
        paSendProgramToFolder ('SP1', 'SubPrograms');
    end_if;
```



5.4.9. paDeleteProgram

Function Delete a program

Syntax `OK := paDeleteProgram (NAME)`

	Parameter	Description
	NAME : STRING;	Name of the program (see notes).
	OK : BOOL;	TRUE if successful.
Description	This function deletes the specified program. It can be used for programs, sub-programs of UDFBS. The <i>NAME</i> parameter can contain '?' and '*' wildchars. For instance, <code>paDeleteProgram ('*')</code> deletes all the POUs of the target project. If the specified program is a SFC program having child programs, its children are deleted as well.	

5.4.10. paCreateProgramFolder

Function Create a folder of programs

Syntax `OK := paCreateProgramFolder (NAME)`

	Parameter	Description
	NAME : STRING;	Name of the folder
	OK : BOOL;	TRUE if successful
Output	NB : DINT; Number of programs Q : STRING; Name of the item selected with ITEM.	

Description This function creates a new folder in the **Programs** tab of the target project workspace.

Attention

This function supports only one level of program folders and cannot be used for nested folders.

Example

```
// create a new program folder
paCreateProgramFolder ('Folder1') then
    // create a new program in LD language
    if paCreateProgram ('Prog1', _LG_LD) then
        // and put it in the folder
        paSendProgramToFolder ('Prog1', 'Folder1');
    end_if;
```



5.4.11. paSendProgramToFolder

Function Move a program to a folder in the workspace

Syntax `OK := paSendProgramToFolder (NAME)`

	Parameter	Description
	<code>PROG : STRING;</code>	Program name
	<code>FOLDER : STRING;</code>	Destination folder
	<code>OK : BOOL;</code>	TRUE if successful
Output	<code>NB : DINT;</code>	Number of programs
	<code>Q : STRING;</code>	Name of the item selected with ITEM

Description This function moves the specified program under the specified folder in the **Programs** tab of the target project workspace. The destination folder must exist before this function is called.

Attention

This function supports only one level of program folders and cannot be used for nested folders.

Example

```
// create a new program folder
paCreateProgramFolder ('Folder1') then
    // create a new program in LD language
    if paCreateProgram ('Prog1', _LG_LD) then
        // and put it in the folder
        paSendProgramToFolder ('Prog1', 'Folder1');
    end_if;
```



5.4.12. paEnumProgramFolder

Function	Enumerate program folders or the target project
Syntax	Inst_paEnumProgramFolder (LOAD, CHECK, ITEM, FILTER)
Input	LOAD : TRUE; If TRUE, open a checklist box to select some of the loaded folders CHECK : BOOL; If TRUE, open a checklist box to select some of the loaded folders ITEM : DINT; Index of the item wanted on input (1 based)
Output	NB : DINT; Number of programs Q : STRING; Name of the item selected with ITEM
Description	This function box is used for enumerating the program folders in the workspace of the target project. You must first call it with LOAD input at <i>TRUE</i> in order to load the list of folders. You get the number of folders in the NB output. Then call it again with LOAD at <i>FALSE</i> and specifying the index of the desired folder in the ITEM input to get the name of the selected item in the Q output. The numbering of items starts at 1. Attention: This function supports only one level of program folders and cannot be used for nested folders. When loading folders (LOAD inputs is <i>TRUE</i>), you can specify a filtering string that may include some '*' or '?' wildchars. After loading, if can call again the block with the CHECK input at <i>TRUE</i> for opening a dialog box where the user can check desired folders. After the box is closed, only checked items remain in the loaded list.

Example

```
// "ENU" is a declared instance of paEnumProgramFolder function block
// load all folders
ENU (TRUE, FALSE, 0, "*");
// open a dialog box for the user to check desired items
ENU (FALSE, TRUE, FALSE, 0, "");
// enumerate checked items
for i := 1 to ENU.NB do
    // select the item "i"
    ENU (FALSE, FALSE, i, "");
    // get the name of the item specified by "i"
    sFolderName := ENU.Q;
end_for;
```



5.4.13. paDeleteProgramFolder

Function Delete a program folder

Syntax `OK := paDeleteProgramFolder (NAME)`

	Parameter	Description
	NAME : STRING;	Name of the folder
	OK : BOOL;	TRUE if successful
Description	This function deletes the specified folder. The <i>NAME</i> parameter can contain '?' and '*' wildchars. For instance, calling <code>paDeleteProgramFolder ("")</code> deletes all the folders of the target project.	
Attention:	This function supports only one level of program folders and cannot be used for nested folders.	



5.5. Declaring data structures

Below are the main services to declaring data structures.

Structure	Explanation
paCreateStruct	Create a data structure
paSetStructComment	Set the description text of a structure

The following are the present data structures

Structure	Explanation
paEnumStruct	Enumerate data structures.
paGetStructDesc	Get Information about a data structure.
paDeleteStruct	Remove a data structure.

5.5.1. Data Structure Functions

Functions	paCreateStruct	Create a data structure
Syntax OK := paCreateStruct (NAME)		
Parameter	NAME : STRING;	Name of the structure
	OK : BOOL;	TRUE if successful
Description This function creates a data structure in the target project.		
Example		

Functions	paSetStructComment	Set the description a structure
Syntax OK := paSetStructComment (NAME, COMM)		
Parameter	NAME : STRING;	Name of the structure
	COMM : STRING;	Description text
	OK : BOOL;	TRUE if successful
Description This function sets the comment text of a data structure.		
Example		



Function Block	paEnumStruct	Enumerate data structures of the target project
Syntax	Inst_paEnumStruct (LOAD, CHECK, ITEM, FILTER)	
Input	LOAD : TRUE; CHECK : BOOL; ITEM : DINT; FILTER : STRING;	If TRUE, load the list of structures If TRUE, open a checklist box to select some of the loaded structures Index of the item wanted on input (1 based) Filtering mask for loading structures
Output	NB : DINT; Q : STRING;	Number of structures Name of the item selected with ITEM
Description	This function block is used for enumerating the data structures of the target project. You must first call it with LOAD input at <i>TRUE</i> in order to load the list of structures. You get the number of structures in the NB output. Then call it again with LOAD at <i>FALSE</i> and specifying the index of the desired structure in the ITEM input to get the name of the selected item in the Q output. The numbering of items starts at 1.	
	When loading structures (LOAD inputs is <i>TRUE</i>), you can specify a filtering string that may include some '*' or '?' wildchars.	
	After loading, if can call again the block with the CHECK input at <i>TRUE</i> for opening a dialog box where the user can check desired structures. After the box is closed, only checked items remain in the loaded list.	
Example	<pre>// "ENU" is a declared instance of paEnumStruct function block // load all structures ENU (TRUE, FALSE, 0, "*"); // open a dialog box for the user to check desired items ENU (FALSE, TRUE, FALSE, 0, ""); // enumerate checked items for i := 1 to ENU.NB do // select the item "i" ENU (FALSE, FALSE, i, ""); // get the name of the item specified by "i" sProgName := ENU.Q; end_for;</pre>	



Function Block paGetStructDesc Get Information about a data structure

Syntax `Inst_paGetStructDesc (NAME)`

Input NAME : STRING; Name of the structure

Output OK : BOOL; TRUE if successful

COMMENT : STRING; Description text

Description This function block is for getting Information about a structure of the target project.

Example

Function Block paDeleteStruct Delete a data structure

Syntax `OK := paDeleteStruct (NAME)`

Parameter NAME : STRING; Name of the structure (see notes)

OK : BOOL; TRUE if successful

Description This function deletes the specified structure. The *NAME* parameter can contain '?' and '*' wildchars. For instance, `paDeleteStruct ("")` deletes all the structures of the target project.

Example



5.6. Declaring Variables

Below are the main services to declaring variables.

paCreateVar	Create a variable
paCreateInputParam	Create an input parameter of a sub-program or UDFB
paCreateOutputParam	Create an output parameter of a sub-program or UDFB
paSetVarXDim(s)	Set dimension(s) of a variable
paSetVarInitValue	Set initial value of a variable
paSetVarComment	Set description text of a variable
paEmbedVarSymbol	Embed the symbol of a variable
paProfileVar	Set variable profile and properties.

Working with existing variables

paEnumVar	Enumerate variables
paGetVarDesc	Get Information about a variable
paDeleteVar	Delete a variable

5.6.1. Naming conventions

In any function where you need to specify a variable by its name, you must specify the group containing the variable. A variable can be Global or I/O, Retain, Local to a POU or can be an item of a data structure.

The convention for these functions is to prefix the variable name by its group name and the '.' separator. If no group is specified, then the variable is considered as Global.

Example

Declaration	Meaning
VarName	A global variable
RETAIN.VarName	A RETAIN variable
%IX1.0	A global I/O channel
ProgName.LocVarName	A variable local to a program
UDFBName.ParamName	A parameter of a UDFB
StructName.Item	An item of a data structure



5.6.2. paCreateVar

Function Create a variable

Syntax `OK := paCreateVar (SNAME, STYPE)`

Parameter	Description
SNAME : STRING;	Name of the new variable and specification of its group (see naming conventions in section 5.6.1 Naming conventions)
STYPE : STRING;	Data type
OK : BOOL;	TRUE if successful

Description This function creates a program with the specified name in the specified group. The STYPE indicates its data type. It can be a type of function block for declaring an instance. In case of a STRING type, the type must be followed by the string length. e.g. 'STRING(255)'

The same function can be used to create an item in a data structure.

Example

```
// create global variable and FB instance
paCreateVar ('Global1', 'BOOL');
paCreateVar ('Timer1', 'TON');

// create a retain variable
paCreateVar ('RETAIN.RetVar1', 'STRING(20)');

// create a variable local to a program
paCreateVar ('Prog1.Local1', 'DINT');

// create an item of a structure
paCreateVar ('Struct1.Item1', 'REAL');
```



5.6.3. paCreateInputParam

Function Create an input parameter of a POU

Syntax `OK := paCreateInputParam (SNAME, STYPE)`

Parameter	Description
<code>SNAME : STRING;</code>	Name of the new variable and specification of its group (see naming conventions in section 5.6.1 Naming conventions)
<code>STYPE : STRING;</code>	Data type
<code>OK : BOOL;</code>	<i>TRUE</i> if successful

Description This function creates an input parameter for the POU (sub-program or UDFB) specified as a prefix in the variable name. The STYPE indicates its data type. In case of a STRING type, the type must be followed by the string length. e.g. 'STRING(255)'.

Example

```
// create a sub program
paCreateSubProgram ('SP', _LG_LD);
// declare parameters
paCreateInputParam ('SP.IN1', 'BOOL');
paCreateInputParam ('SP.IN2', 'DINT');
paCreateOutputParam ('SP.Q', 'BOOL');
```



5.6.4. paCreateOutputParam

Function Create an output parameter of a POU

Syntax `OK := paCreateOutputParam (SNAME, STYPE)`

Parameter	Description
SNAME : STRING;	Name of the new variable in form 'POUName.VarName' (see naming conventions in section 5.6.1 Naming conventions)
STYPE : STRING;	Data type
OK : BOOL;	TRUE if successful

Description This function creates an output parameter for the POU (sub-program or UDFB) specified as a prefix in the variable name. The STYPE indicates its data type. In case of a STRING type, the type must be followed by the string length. e.g. 'STRING(255)'.

Example

```
// create a sub program
paCreateSubProgram ('SP', _LG_LD);
// declare parameters
paCreateInputParam ('SP.IN1', 'BOOL');
paCreateInputParam ('SP.IN2', 'DINT');
paCreateOutputParam ('SP.Q', 'BOOL');
```



5.6.5. paSetVarDim

Function paSetVar1Dim / paSetVar2Dims / paSetVar3Dims - Set the demension(s) of a variable (array)

Syntax

```
OK := paSetVarDim (NAME, DIM) // 1 dimension array
OK := paSetVar2Dims (NAME, DIMHIGH, DIMLOW) // 2 dimension array
OK := paSetVar3Dims (NAME, DIMHIGH, DIMMEDIUM, DIMLOW) // 3 dimension array
```

Parameter	Description
NAME : STRING;	Name of the variable and specification of its array (see naming conventions in section 5.6.1 Naming conventions)
DIMxxx : DINT;	Dimension(s) of the array
OK : BOOL;	TRUE if successful

Description This function sets the dimension of a declared variable, in order to declare an array. The same function can be used for an item in a data structure.

Example

```
// declare Array1 : array [0..9] of BOOL
paCreateVar ('Array1', 'BOOL');
paSetVarDim ('Array1', 10);

// declare Array2 : array [0..9,0..4] of BOOL
paCreateVar ('Array2', 'BOOL');
paSetVar2Dims ('Array2', 10, 5);

// declare Array2 : array [0..9,0..4,0..19] of BOOL
paCreateVar ('Array3', 'BOOL');
paSetVar3Dims ('Array3', 10, 5, 20);
```



5.6.6. paSetVarInitValue

Function Set the initial value of a variable

Syntax `OK := paSetVarInitValue (NAME, VALUE)`

Parameter	Description
NAME : STRING;	Name of the variable and specification of its group (see naming conventions in section 5.6.1 Naming conventions)
VALUE : STRING;	Initial value in IEC 61131-3 syntax
OK : BOOL;	TRUE if successful

Description This function sets the initial value of a declared variable. The same function can be used for an item in a data structure.

Example

```
// declare Integer1 : DINT := 1234;
paCreateVar ('Integer1', 'BOOL');
paSetVarInitValue ('Integer1', '1234');

// declare Array1 : array [0..3] of BOOL := 1, 2, 3, 4;
paCreateVar ('Array1', 'BOOL');
paSetVarDim ('Array1', 4);
paSetVarInitValue ('Integer1', '1,2,3,4');
```

5.6.7. paSetVarComment

Function Set the description a variable

Syntax `OK := paSetVarComment (NAME, COMM, TAG)`

Parameter	Description
NAME : STRING;	Name of the variable and specification of its group (see naming conventions in section 5.6.1 Naming conventions)
COMM : STRING;	Description text
TAG : STRING;	Short description text
OK : BOOL;	TRUE if successful

Description This function sets the comment text of a declared variable. The same function can be for an item in a data structure.

Example



5.6.8. paEmbedVarSymbol

Function Embed the symbol of a variable

Syntax `OK := paEmbedVarSymbol (NAME, SYB)`

Parameter	Description
NAME : STRING;	Name of the variable and specification of its group (see naming conventions in section 5.6.1 Naming conventions)
SYB : BOOL;	TRUE if the variable symbol must be embedded
OK : BOOL;	TRUE if successful

Description This function specifies if the symbol of a declared variable must be embedded. The same function can be for an item in a data structure.

Example

5.6.9. paProfileVar

Function Defines the profile and embedded properties of a variable

Syntax `OK := paProfileVar (PROFILE, PROPERTIES)`

Parameter	Description
NAME : STRING;	Name of the variable and specification of its group (see naming conventions in section 5.6.1 Naming conventions)
PROFILE : STRING;	Name of the profile
PROPERTIES : STRING;	Embedded properties attached to the profile (if any)
OK : BOOL;	TRUE if successful

Description This function specifies the profile and the corresponding embedded properties for a declared variable. Properties are written in form:

```
prop=value,prop=value,...
```

Example

```
// declare the variable
paCreateVar ('F1', 'BOOL');
paProfileVar ('K1', '1234');
// declare Array1 : array [0..3] of BOOL := 1, 2, 3, 4;
paCreateVar ('Array1', 'FKey', 'Key=1');
```



5.6.10. paEnumVar

Function Enumerate variables of a group in the target project

Syntax **Inst_paEnumVar (LOAD, CHECK, ITEM, FILTER)**

	Parameter	Description
Input	LOAD : TRUE;	If TRUE, load the list of variables
	CHECK : BOOL;	If TRUE, open a checklist box to select some of the loaded variables
	ITEM : DINT;	Index of the item wanted on input (1 based)
	FILTER : STRING;	Group specification and filtering mask for loading variables
Output	NB : DINT;	Number of variables
	Q : STRING;	Name of the item selected with "ITEM"
Description	This function block is used for enumerating the variables of a group in the target project. First call it with LOAD input at <i>TRUE</i> in order to load the list of variables. You obtain the number of programs in the NB output. Then call it again with LOAD at <i>FALSE</i> and specifying the index of the desired variable in the ITEM input to get the name of the selected item in the Q output. The numbering of items starts at 1.	
	When loading programs (LOAD inputs is <i>TRUE</i>), you must specify the group in the FILTER parameter, possibly followed by the '.' separator and a filtering string including some '*' or '?' wildchars. No group name is required for global variables.	
	After loading, if can call again the block with the CHECK input at <i>TRUE</i> for opening a dialog box where the user can check desired variables. After the box is closed, only checked items remain in the loaded list.	

Example

```
// "ENU" is a declared instance of paEnumVar function block
// load all RETAIN variables
ENU (TRUE, FALSE, 0, 'RETAIN.*');
// open a dialog box for the user to check desired items
ENU (FALSE, TRUE, FALSE, 0, '');
// enumerate checked items
for i := 1 to ENU.NB do
    // select the item "i"
    ENU (FALSE, FALSE, i, '');
    // get the name of the item specified by "i"
    sVarName := ENU.Q;
end_for;
```



5.6.11. paGetVarDesc

Function Get Information about a variable

Syntax `Inst_paGetVarDesc (NAME)`

	Parameter	Description
Input	NAME : STRING;	Name of the variable and specification of ots group (see naming conventions)
Output	OK : BOOL;	<i>TRUE</i> if successful
	INPARAM : BOOL;	<i>TRUE</i> if it is an input parameter
	OUTPARAM : BOOL;	<i>TRUE</i> if it is an output parameter
	DATATYPE : STRING;	Data type
	DIMTOTAL : DINT;	Total number of items for an array
	DIMHIGH : DINT;	Highest dimension (0 for 1 or 2 dimension arrays)
	DIMMEDIUM : DINT;	Medium dimension (0 for 1 dimension arrays)
	DIMLOW : DINT;	Lowest dimension
	INIT : STRING;	Initial value in IEC syntax
	SYB : BOOL;	<i>TRUE</i> if the symbol of the variable is embedded
	PROFILE : STRING;	Name of the attached profile
	PROPS : STRING;	Embedded properties attached to the profile
	COMMENT : STRING;	Description text
	TAG : STRING;	Short description text
Description	This function block is for getting Information about a declared variable. The same block can be used for an item in a data structure.	

Example



5.6.12. paDeleteVar

Function Delete a variable

Syntax `OK := paDeleteVar (NAME)`

Parameter	Description
NAME : STRING;	Name of the variable and specification of its group (see naming conventions)
OK : BOOL;	TRUE if successful

Description This function deletes the specified variable. The same function can be used for an item in a data structure.

The name of the variable may contain '?' and '*' wildchars, but the group name cannot. For instance, `paDeleteVar ('RETAIIN.*')` deletes all the retain variables.

Example



5.7. Declaring I/O Boards

Below are the main services to declaring I/O Boards.

paCreateIBoard	Create an instance of an IO device
paSetIBoardComment	Set description text for an IO device
paSetIBoardParam	Set a parameter of an IO board
paSetIOAlias	Define an alias for a channel of an IO board

Working with existing I/O boards

paEnumIBoards	Enumerate IO boards.
paGetIBoardDesc	GetDescription of an IO board.
paDeleteIBoard	Delete an instance of an IO device.
paDeleteAllIBoards	Delete all IO devices.

5.7.1. I/O Board Functions

Functions	paCreateIBoard	Create an instance of an IO device
Syntax <code>OK := paCreateIBoard (SLOT, DEVTYPE)</code>		
Parameter	SLOT : DINT;	Slot number (0..255)
	DEVTYPE : STRING;	Name of the type of IO device
	OK : BOOL;	TRUE if successful
Description	This function creates an instance of the specified IO device at the specified slot number.	

Example

Functions	paSetIBoardComment	Set the description an IO device
Syntax <code>OK := paSetIBoardComment (SLOT, COMM)</code>		
Parameter	SLOT : DINT;	Slot number (0..255)
	COMM : STRING;	Description text
	OK : BOOL;	True if successful
Description	This function sets the comment text of the specified instance of an IO device.	

Example



Functions **paSetIOBoardParam** Set the value of a parameter of an IO device

Syntax **OK := paSetIOBoardComment (SLOT, COMM)**

Parameter **BOARD : STRING;** Board name (e.g. '%QX0')

PARAM : STRING; Name of the parameter

VALUE : STRING; Value of the parameter

OK : BOOL; *TRUE* if successful

Description This function sets the value of an IO board parameter. Refer to OEM instruction for details about available parameters.

Example

Functions **paSetIOAlias** Define an alias for an IO channel

Syntax **OK := paSetIOBoardComment (SLOT, COMM)**

Parameter **CHANNEL : STRING;** Channel name (e.g. '%QX0.1')

ALIAS : STRING; Readable name to be used as an alias for the channel

OK : BOOL; *TRUE* if successful

Description This function defines a readable name for the specified IO channel. The alias can then be used in place of the "%" name in programs.

Example



Functions **paSetIOAlias**

Define an alias for an IO channel

Syntax **OK := paSetIOBoardComment (SLOT, COMM)**

Parameter CHANNEL : STRING; Channel name (e.g. '%QX0.1')

ALIAS : STRING; Readable name to be used as an alias for the channel

OK : BOOL; *TRUE* if successful

Description This function defines a readable name for the specified IO channel. The alias can then be used in place of the "%" name in programs.

Example



5.7.2. paEnumIOBoards

Function Enumerate IO boards of the target project

Syntax `Inst_paEnumIOBoard (LOAD, CHECK, ITEM, FILTER)`

	Parameter	Description
Input	LOAD : TRUE;	If TRUE, load the list of boards
	CHECK : BOOL;	If TRUE, open a checklist box to select some of the loaded boards
	ITEM : DINT;	Index of the item wanted on input (1 based)
Output	NB : DINT;	Number of boards
	Q : STRING;	Name of the item selected with "ITEM"
Description	This function block is used for enumerating the IO boards of the target project. You must first call it with LOAD input at <i>TRUE</i> in order to load the list of IO boards. You get the number of boards in the NB output. Then call it again with LOAD at <i>FALSE</i> and specifying the index of the desired structure in the ITEM input to get the name of the selected item in the Q output. The numbering of items starts at 1.	
	For complex IO devices, only the child boards of the device are listed. The complex device itself does not occur in the list.	
	After loading, if can call again the block with the CHECK input at <i>TRUE</i> for opening a dialog box where the user can check desired boards. After the box is closed, only checked items remain in the loaded list.	
Example	<pre>// "ENU" is a declared instance of paEnumIOBoard function block // load all boards ENU(TRUE, FALSE, 0); // open a dialog box for the user to check desired items ENU(FALSE, TRUE, FALSE, 0); // enumerate checked items for i := 1 to ENU.NB do // select the item "i" ENU(FALSE, FALSE, i); // get the name of the item specified by "i" sBoardName := ENU.Q; end_for;</pre>	



5.7.3. paGetIOBoardDesc

Function Get Information about an IO board

Syntax `Inst_paGetIOBoardDesc (NAME)`

	Parameter	Description
Input	NAME : STRING;	Name of the board (e.g. '%QX0')
Output	OK : BOOL;	TRUE if successful
	NBCHANNEL : DINT;	Number of channels
	DEVNAME : STRING;	IO Device name
	DEVGROUP : STRING;	In case of a complex IO device, name of the group within the device
	COMMENT : STRING;	Description text

Description This function block is for getting Information about an IO board of the target project.

Example

5.7.4. paDeleteIOBoard

Function Delete an instance of an IO device

Syntax `OK := paDeleteIOBoard (SLOT)`

Parameter	SLOT : DINT;	Slot number (0..255)
	OK : BOOL;	TRUE if successful

Description This function deletes the specified instance of an IO device. In case of a complex device, the whole device (with all its groups) is removed.

Example

5.7.5. paDeleteAllIOBoards

Function Delete all instances of IO devices

Syntax `OK := paDeleteIOBoard (SLOT)`

Parameter	OK : BOOL;	TRUE if successful
------------------	------------	--------------------

Description This function deletes all the existing instances of any IO device.

Example



Chapter 6: Generating documents

6.1. Common File Services

Below are the main services for managing files when generating documents in the target project.

paFileOpenWrite	Open any file in the target project for writing
paFileOpenWriteProgramSrc	Open the source file of a POU for writing
paFileOpenWriteProgramDef	Open the definitions file of a POU for writing
paFileClose	Close an open file

6.1.1. paFileOpenWrite

Function Open any file in the target project for writing

Syntax `FID := paFileOpenWrite (PREFIX, SUFFIX)`

Parameter PREFIX : STRING; File prefix

 SUFFIX : STRING; File suffix

 FID : DINT; File identifier or 0 if fail

Description This function opens a target project file for writing. It should normally not be used as dedicated file open functions are available for most of possible project documents.

This function removes the contents of the file if it already exists.

When written, the file must be closed by calling the paFileClose function.

Example



6.1.2. paFileOpenWriteProgramSrc

Function Open a POU source file for writing

Syntax `FID := paFileOpenWriteProgramSrc (NAME)`

Parameter NAME : STRING; Program name

FID : DINT; File identifier or 0 if fail

Description This function opens a target project file for writing. It should normally open for writing the file that contains the source code of a program, sub-program or UDFB. Then you should use the appropriate file writing functions according to the language of the specified POU.

This function removes the contents of the file if it already exists.

When written, the file must be closed by calling the paFileClose function.

Example

6.1.3. paFileOpenWriteProgramDef

Function Open a POU definitions file for writing

Syntax `FID := paFileOpenWriteProgramDef (NAME)`

Parameter NAME : STRING; Program name

FID : DINT; File identifier or 0 if fail

Description This function opens for writing the file that contains the local definitions of a program, sub-program or UDFB. Then you should use the text file writing functions to fill the file.

This function removes the contents of the file if it already exists.

When written, the file must be closed by calling the paFileClose function.

Example



6.1.4. paFileClose

Function Close an open file

Syntax `OK := paFileClose (FID)`

Parameter `FID : DINT;` File identifier returned by any file open function.

Description This function closes a file open by any of available file open functions.

Example



6.2. Text File Writing Services

Below are the main services for writing in text files.

paFTextEol Write end of line characters to a text file

paFTextLine Write a line in a text file

paFTextString Write a string in a text file

6.2.1. Text File Writing Functions

Functions paFTextEol Write end of line characters in a text file

Syntax **OK := paFTextEol (FID)**

Parameter FID : DINT; File identifier returned by the file open function

Description This function writes end of line characters in a text file open for writing.

Example

Functions paFTextLine Write a line in a text file

Syntax **OK := paFTextLine (FID, TEXT)**

Parameter FID : DINT; File identifier returned by the file open function

TEXT : STRING; Text to be written

Description This function writes the string specified by the TEXT parameter plus end of line characters in a text file open for writing.

Example

Functions paFTextString Write a string in a text file

Syntax **OK := paFTextEol (FID, TEXT)**

Parameter FID : DINT; File identifier returned by the file open function

TEXT : STRING; Text to be written

Description This function writes the string specified by the TEXT parameter in a text file open for writing.

Example



6.3. IEC Source Code File Writing Services

Below are the main services for writing source code in IEC language in POU source files. Files are open using the paFileOpenWriteSrc function. When complete, they must be closed by calling the paFileClose function.

There are various set of available functions depending on the programming language:

Structured Text or Instruction List	Text file writing functions
Function Block Diagram	Functions for FBD sequential writing
Ladder Diagram	Functions for LD sequential writing
Sequential Function Chart	Functions for SFC sequential

6.3.1. ST / IL Text

Below are the main services for writing in text files.

paFTextEol	Write end of line characters to a text file
paFTextLine	Write a line in a text file
paFTextString	Write a string in a text file

6.3.2. ST/IL Text Functions

Functions	paFTextEol	Write end of line characters in a text file
	Syntax	OK := paFTextEol (FID)
	Parameter	FID : DINT; File identifier returned by the file open function.
Description This function writes end of line characters in a text file open for writing.		
Example		
Functions	paFTextLine	Write a line in a text file
	Syntax	OK := paFTextLine (FID, TEXT)
	Parameter	FID : DINT; File identifier returned by the file open function
		TEXT : STRING; Text to be written
	Description	This function writes the string specified by the TEXT parameter plus end of line characters in a text file open for writing.
Example		



Functions **paFTextString**

Write a string in a text file

Syntax **OK := paFTextEol (FID, TEXT)**

Parameter **FID : DINT;** File identifier returned by the file open function

TEXT : STRING; Text to be written

Description This function writes the string specified by the TEXT parameter in a text file open for writing.

Example



6.3.3. FBD Sequential File Writing Services

The functions below enable you to write sequentially (from the top to the bottom) the source code of a program written in FBD language. It has limited features (it is not possible to link 2 blocks together) but provides a very easy way to generate FBD as you do not need to care with the position of the elements in the diagram. Files are open using the paFileOpenWriteSrc function. When complete, they must be closed by calling the paFileClose function.

Use the following functions for inserting objects sequentially:

paFbdsBreak	Insert a network break
paFbdsComment	Insert a multiline comment block
paFbdsLabel	Insert a label
paFbdsJump	Insert a unconditional jump
paFbdsReturn	Insert a unconditional <RETURN> statement
paFbdsBlock	Insert a block.

Just after inserting a block, you then can define its inputs and outputs:

paFbdsInputVar	Connect a variable on input of the last block
paFbdsOutputVar	Connect a variable on output of the last block
paFbdsOutputJump	Connect a jump statement on output of the last block
paFbdsOutputReturn	Connect a <RETURN> statement on output of the last block



6.3.4. FBD Sequential File Writing functions

Functions	paFbdsBreak	add a network break
Syntax <code>OK := paFbdsBreak (FID, TEXT)</code>		
Parameter	FID : DINT;	File identifier answered by paFileOpenWriteSrc
	TEXT : STRING;	Text written on the network break
	OK : BOOL;	TRUE if successful
Description	This function adds a network break under the last added item in a FBD source file open for writing.	

Example

Functions	paFbdsComment	add a comment block
Syntax <code>OK := paFbdsComment (FID, TEXT, HEIGHT)</code>		
Parameter	FID : DINT;	File identifier answered by paFileOpenWriteSrc
	TEXT : STRING;	Comment text
	HEIGHT ; DINT;	Height of the comment block (in diagram grid unit)
	OK : BOOL;	TRUE if successful
Description	This function adds a multiline comment block under the last added item in a FBD source file open for writing. You can use the '\$N' sequence in the comment text to specify an end of line.	

Example

Functions	paFbdsLabel	add a label
Syntax <code>OK := paFbdsLabel (FID, LABEL)</code>		
Parameter	FID : DINT;	File identifier answered by paFileOpenWriteSrc
	LABEL : STRING;	Label name
	OK : BOOL;	TRUE if successful
Description	This function adds a label under the last added item in a FBD source file open for writing. The label is placed on the left side of the diagram.	

Example

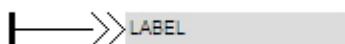


Functions **paFbdsJump** add a unconditional jump

Syntax **OK := paFbdsLabel (FID, LABEL)**

Parameter FID : DINT; File identifier answered by paFileOpenWriteSrc
LABEL : STRING; Target label name
OK : BOOL; TRUE if successful

Description This function adds a jump instruction under the last added item in a FBD source file open for writing. It is an unconditional jump instruction. The jump symbol is placed on the left side of the diagram, connected to a LD left power rail (always true):



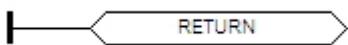
Example

Functions **paFbdsReturn** add a unconditional <return> symbol

Syntax **OK := paFbdsReturn (FID)**

Parameter FID : DINT; File identifier answered by paFileOpenWriteSrc
OK : BOOL; TRUE if successful

Description This function adds a jump instruction under the last added item in a FBD source file open for writing. It is an unconditional jump instruction. The jump symbol is placed on the left side of the diagram, connected to a LD left power rail (always true):



Example



Functions **paFbdsBlock** add a block

Syntax **OK := paFbdsBlock (FID, NAME, INSTANCE)**

Parameter FID : DINT; File identifier answered by paFileOpenWriteSrc
NAME : STRING; Block name
INSTANCE : STRING; Instance name in case of a function block
OK : BOOL; TRUE if successful

Description This function adds a block symbol under the last added item in a FBD source file open for writing. It can be a basic operator, a function or a function block. The name of the instance must be specified in case of a function block.

Immediately after calling this function, use the FbdsInput... and FbdsOutput... functions in order to connect the input and output pins of the block.

Example

Functions **paFbdsInputVar** connect an input of the last added block

Syntax **OK := paFbdsInputVar (FID, NAME, PIN, NEGATE)**

Parameter FID : DINT; File identifier answered by paFileOpenWriteSrc
NAME : STRING; Variable name or constant expression
PIN : DINT; Index of the input pin of the block (the first input pin is 1)
NEGATE : BOOL; If TRUE, the connection between the variable and the block has a Boolean negation
OK : BOOL; TRUE if successful

Description This function connects a variable box on input of the last block added by the paFbdsBlock function, and must be called immediately after.

Example



Functions	paFbdsOutputVar	connect an output of the last added block
	Syntax	OK := paFbdsOutputVar (FID, NAME, PIN, NEGATE)
	Parameter	FID : DINT; File identifier answered by paFileOpenWriteSrc
		NAME : STRING; Variable name
		PIN : DINT; Index of the output pin of the block (the first output pin is 1)
		NEGATE : BOOL; If TRUE, the connection between the block and the variable has a Boolean negation
		OK : BOOL; TRUE if successful
	Description	This function connects a variable box on output of the last block added by the paFbdsBlock (Error! Bookmark not defined.) function, and must be called immediately after.

Example

Functions	paFbdsOutputJump	connect a jump symbol on output of the last added block
	Syntax	OK := paFbdsOutputJump (FID, LABEL, PIN, NEGATE)
	Parameter	FID : DINT; File identifier answered by paFileOpenWriteSrc
		LABEL : STRING; Target label name
		PIN : DINT; Index of the output pin of the block (the first output pin is 1)
		NEGATE : BOOL; If TRUE, the connection between the block and the variable has a Boolean negation
		OK : BOOL; TRUE if successful
	Description	This function connects a jump instruction on output of the last block added by the paFbdsBlock (Error! Bookmark not defined.) function, and must be called immediately after.

Example



Functions **paFbdsOutputReturn** connect a <return> jump symbol on output of the last added block

Syntax `OK := paFbdsOutputReturn (FID, PIN, NEGATE)`

Parameter `FID : DINT;` File identifier answered by `paFileOpenWriteSrc`

`PIN : DINT;` Index of the output pin of the block (the first output pin is 1)

`NEGATE : BOOL;` If TRUE, the connection between the block and the variable has a Boolean negation

`OK : BOOL;` TRUE if successful

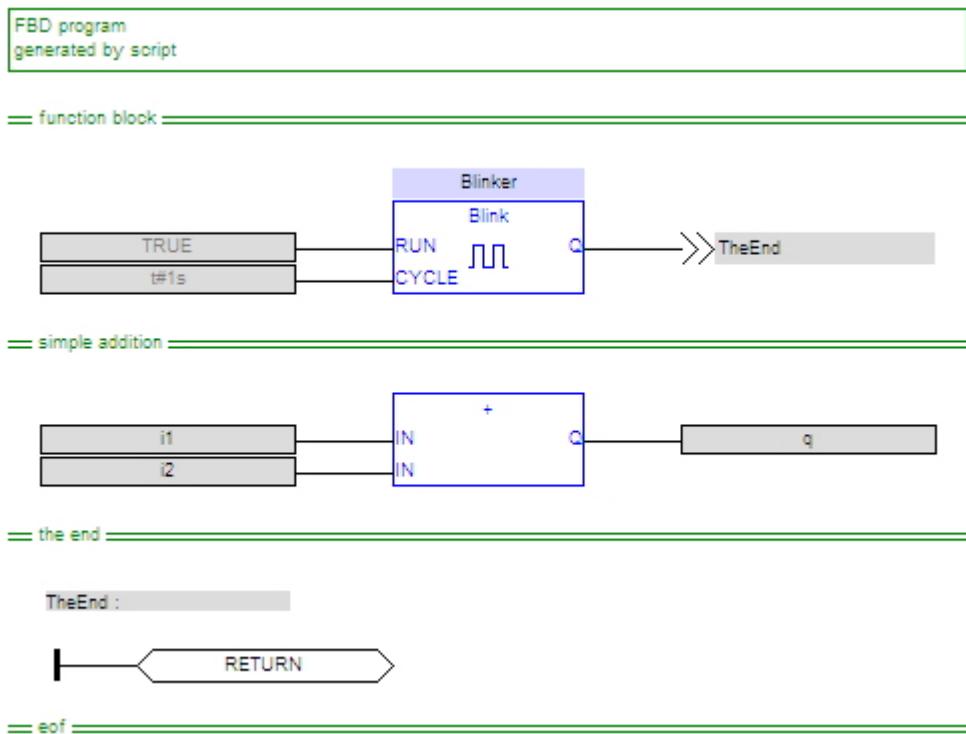
Description This function connects a <RETURN> instruction on output of the last block added by the `paFbdsBlock` (**Error! Bookmark not defined.**) function, and must be called immediately after.

Example Below is an Example of script that generates a FBD file in the target project:

```
// make FBD code - sequential writing
f := paFileOpenWriteProgramSrc ('ProgFBD');
if f <> 0 then
    // add a comment block (2 lines of text)
    paFbdsComment (f, 'FBD program$Ngenerated by script', 2);
    // add a function block and its inputs and outputs
    paFbdsBreak (f, 'function block');
    paFbdsBlock (f, 'Blink', 'Blinker');
    paFbdsInputVar (f, 'TRUE', 1, FALSE);
    paFbdsInputVar (f, 't#1s', 2, FALSE);
    paFbdsOutputJump (f, 'TheEnd', 1, FALSE);
    // add a function and its inputs and outputs
    paFbdsBreak (f, 'simple addition');
    paFbdsBlock (f, '+', '');
    paFbdsInputVar (f, 'i1', 1, FALSE);
    paFbdsInputVar (f, 'i2', 2, FALSE);
    paFbdsOutputVar (f, 'q', 1, FALSE);
    // add a label and an unconditional return
    instruction
    paFbdsBreak (f, 'the end');
    paFbdsLabel (f, 'TheEnd');
    paFbdsReturn (f);
    // add a break at the end of the diagram
    paFbdsBreak (f, 'eof');
    // close the file
    paFileClose (f);
end_if,
```



Here is the FBD program generated by the script:





6.3.5. LD File Writing Services

The functions below enable you to write sequentially (from the top to the bottom) the source code of a program written in LD language. The program is written rung after rung. Items are placed sequentially on the rung from the left to the right. Files are open using the paFileOpenWriteSrc function. When complete, they must be closed by calling the paFileClose (**Error! Bookmark not defined.**) function.

Use the following functions for inserting rungs in the diagram:

paFLdCommentLine	Add a comment line (network break)
paFLdStartRung	Start a new rung
paFLdEndRung	Terminate the rung
paFLdDivergence	Start an <i>OR</i> divergence on the current rung
paFLdConvergence	Terminate an <i>OR</i> divergence on the current rung
paFLdNextBranch	Add a parallel branch to the current divergence

Use the following functions for adding items on the current rung:

paFLdContact	Add a contact
paFLdCoil	Add a coil
paFLdJump	Add a jump instruction
paFLdReturn	Add a <return> jump instruction
paFLdBlock	Add a block
paFLdHorzSegment	Add a horizontal segment line



6.3.6. LD File Writing Functions

Functions **paFLdCommentLine** add a network break

Syntax **OK := paFLdCommentLine (FID, TEXT)**

Parameter **FID : DINT;** File identifier answered by
paFileOpenWriteSrcError! **Bookmark not defined.**

TEXT : STRING; Text written on the network break

OK : BOOL; TRUE if successful

Description This function adds a comment line (network break) in between rungs

Example

Functions **paFLdStartRung** start a new rung

Syntax **OK := paFLdStartRung (FID, LABEL)**

Parameter **FID : DINT;** File identifier answered by
paFileOpenWriteSrcError! **Bookmark not defined.**

LABEL : STRING; Rung label name (empty string if no label defined)

OK : BOOL; TRUE if successful

Description This function starts a new rung. After calling this function, you can call other functions to add items on the rung from the left to the right. When complete, you need to call the paFLdEndRung function to terminate the rung.

Example

Functions **paFLdEndRung** terminates the rung

Syntax **OK := paFLdEndRung (FID)**

Parameter **FID : DINT;** File identifier answered by
paFileOpenWriteSrcError! **Bookmark not defined.**

FID : DINT; File identifier answered by
paFileOpenWriteSrcError! **Bookmark not defined.**

OK : BOOL; TRUE if successful



Description This function terminates the current rung. You need to call it before starting another rung or adding a comment line.

Example

Functions **paFLdDivergence** starts a divergence

Syntax **OK := paFLdDivergence (FID)**

Parameter FID : DINT; File identifier answered by paFileOpenWriteSrc **Error! Bookmark not defined.**

OK : BOOL; TRUE if successful

Description This function starts a *OR* divergence on the current rung. After calling this function, items can be added on the first (top) branch of the divergence. Parallel branches can be added by calling the paFLdNextBranch function. When complete, you need to call the paFLdConvergence function to terminate the divergence.

Example

Functions **paFLdConvergence** terminates a divergence

Syntax **OK := paFLdConvergence (FID)**

Parameter FID : DINT; File identifier answered by paFileOpenWriteSrc **Error! Bookmark not defined.**

OK : BOOL; TRUE if successful

Description This function terminates the current *OR* divergence. After calling this function, items can be added after the divergence on the parent rung.

Example

Functions **paFLdNextBranch** add a branch to a divergence

Syntax **OK := paFLdNextBranch (FID)**

Parameter FID : DINT; File identifier answered by paFileOpenWriteSrc **Error! Bookmark not defined.**

OK : BOOL; TRUE if successful

Description This function starts a new parallel branch in the current *OR* divergence. After calling this function, items can be added on the new branch.



Example



Functions paFLdContact add a contact

Syntax `OK := paFLdContact (FID, TYP, NAME)`

Parameter FID : DINT; File identifier answered by
paFileOpenWriteSrc **Error! Bookmark not defined.**

TYP : STRING; Type of contact (see notes)

NAME : STRING; Name of the variable attached to the contact

OK : BOOL; TRUE if successful

Description This function adds a contact on the current rung. The following values are available for the TYP parameter:

Value	Description
_LD_DIR	Normal contact
_LD_INV	Negated contact
_LD_P	Positive pulse contact
_LD_N	Negative pulse contact

Example



Functions **paFLdCoil** add a coil

Syntax **OK := paFLdCoil (FID, TYP, NAME)**

Parameter FID : DINT; File identifier answered by
paFileOpenWriteSrc **Error! Bookmark not defined.**

TYP : STRING; Type of coil (see notes)

NAME : STRING; Name of the variable attached to the coil

OK : BOOL; TRUE if successful

Description This function adds a coil on the current rung. The following values are available for the TYP parameter:

Value	Description
_LD_DIR	Normal coil.
_LD_INV	Negated coil.
_LD_P	Positive pulse coil.
_LD_N	Negative pulse coil.
_LD_SET	Set coil.
_LD_RESET	Reset coil.

Example

Functions **paFLdJump** add a jump symbol

Syntax **OK := paFLdJump (FID, LABEL)**

Parameter FID : DINT; File identifier answered by
paFileOpenWriteSrc **Error! Bookmark not defined.**

LABEL : STRING; Target label name

OK : BOOL; TRUE if successful

Description This function adds a jump instruction on the current rung.

Example



Functions paFLdReturn add a <return> jump symbol

Syntax `OK := paFLdReturn (FID)`

Parameter `FID : DINT;` File identifier answered by
paFileOpenWriteSrc **Error! Bookmark not defined.**

`OK : BOOL;` TRUE if successful

Description This function adds a <RETURN> jump instruction on the current rung.

Example

Functions paFLdBlock add a block

Syntax `OK := paFLdBlock (FID, NAME, INSTANCE, INPUTS, OUTPUTS)`

Parameter `FID : DINT;` File identifier answered by
paFileOpenWriteSrc **Error! Bookmark not defined.**

`NAME : STRING;` Type of block.

`INSTANCE : STRING;` Name of the instance in case of a function block.

`INPUTS : STRING;` List of input values (see notes).

`OUTPUTS : STRING;` List of output values (see notes).

Description This function adds a block on the current rung. In case of a function block, you must specify the name of the instance. Blocks are connected to the rung by their first input and output. Name of additional inputs and outputs are passed in strings separated by comas. Below is the Example of an "addition" block with 2 additional inputs calle 'i1' and 'i2'and 1 additional output calle 'q':

Example

```
paFLdStartRung (f, "");  
paFLdBlock (f, '+', "", 'i1,i2', 'q');  
paFLdCoil (f, _LD_DIR, "");  
paFLdEndRung (f);
```



Functions **paFLdHorzSegment** add a horizontal segment

Syntax **OK := paFLdHorzSegment (FID)**

Parameter **FID : DINT;** File identifier answered by
paFileOpenWriteSrc **Error! Bookmark not defined.**

OK : BOOL; TRUE if successful

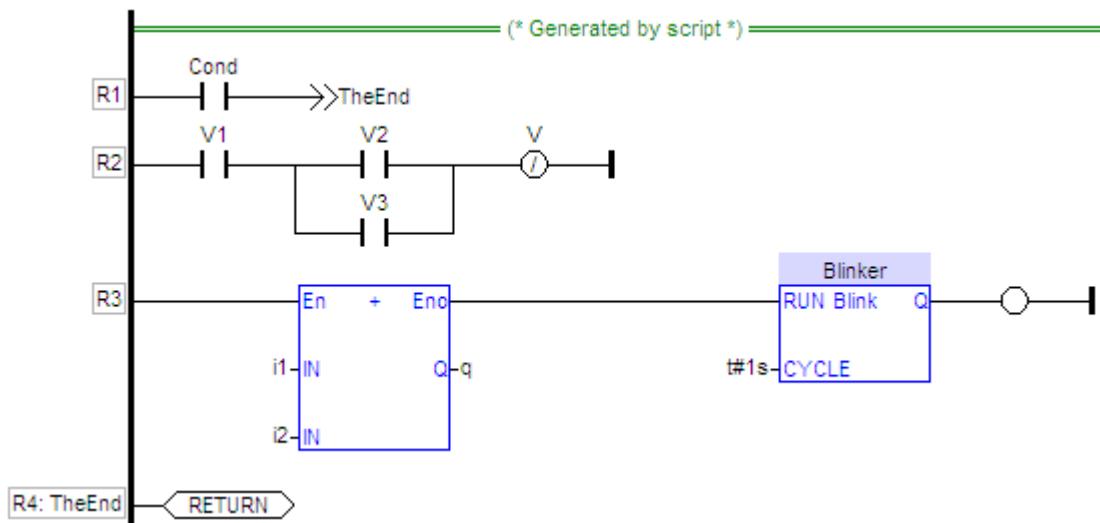
Description This function adds a horizontal line segment on the current rung.
This can be used for aligning items vertically in the diagram.

Example Below is an Example of script that generates a LD file in the target project:

```
// make LD code
f := paFileOpenWriteProgramSrc ('ProgLD');
if f <> 0 then
    // comment line
    paFLdCommentLine (f, 'Generated by script');
    // a simple rung finishing with a jump
    paFLdStartRung (f, '');
    paFLdContact (f, _LD_DIR, 'Cond');
    paFLdJump (f, 'TheEnd');
    paFLdEndRung (f);
    // a rung with contacts and coils
    paFLdStartRung (f, '');
    paFLdContact (f, _LD_DIR, 'V1');
    paFLdDivergence (f);
    paFLdContact (f, _LD_DIR, 'V2');
    paFLdNextBranch (f);
    paFLdContact (f, _LD_DIR, 'V3');
    paFLdConvergence (f);
    paFLdCoil (f, _LD_INV, 'V');
    paFLdEndRung (f);
    // a rung with blocks
    paFLdStartRung (f, '');
    paFLdBlock (f, '+', '', 'i1,i2', 'q');
    paFLdBlock (f, 'Blink', 'Blinker', 't#1s', '');
    paFLdCoil (f, _LD_DIR, '');
    paFLdEndRung (f);
    // a rung with a label and a RETURN instruction
    paFLdStartRung (f, 'TheEnd');
    paFLdReturn (f);
    paFLdEndRung (f);
    // close the file
    paFileClose (f);
end_if;
```



Here is the LD program generated by the script:



6.3.7. SFC File Writing Services

The functions below enable you to write sequentially (from the top to the bottom) the source code of a program written in SFC language. The program is written sequentially, from the top to the bottom and branch per branch. Files are open using the `paFileOpenWriteSrcError! Bookmark not defined.` function. When complete, they must be closed by calling the `paFileCloseError! Bookmark not defined.` function.

6.3.8. SFC File Writing Functions

Use the following functions for inserting SFC items in the chart:



Macro-steps are not supported.

Functions	paSfcIStep	add an initial step
	Syntax	OK := paSfcIStep (FID, REF)
	Parameter	FID : DINT; File identifier answered by paFileOpenWriteSrcError! Bookmark not defined.
		REF : DINT; Reference number of the step
		OK : BOOL; TRUE if successful
	Description	This function adds an initial step at the end of the current branch.
	Example	
Functions	paSfcStep	add a step
	Syntax	OK := paSfcStep (FID, REF)
	Parameter	FID : DINT; File identifier answered by paFileOpenWriteSrcError! Bookmark not defined.
		REF : DINT; Reference number of the step
		OK : BOOL; TRUE if successful
	Description	This function adds a step at the end of the current branch.
	Example	



Functions	paSfcTrans	add a transition
Syntax	OK := paSfcTrans (FID, REF)	
Parameter	FID : DINT;	File identifier answered by paFileOpenWriteSrcError! Bookmark not defined.
	REF : DINT;	Reference number of the transition
	OK : BOOL;	TRUE if successful
Description	This function adds a transition at the end of the current branch.	

Example

Functions	paSfcJump	add a jump to a step
Syntax	OK := paSfcJump (FID, REF)	
Parameter	FID : DINT;	File identifier answered by paFileOpenWriteSrcError! Bookmark not defined.
	REF : DINT;	Reference number of the target step
	OK : BOOL;	TRUE if successful
Description	This function adds a jump to a step at the end of the current branch.	

Example

Functions	paSfcDiv	start a divergence
Syntax	OK := paSfcDiv (FID)	
Parameter	FID : DINT;	File identifier answered by paFileOpenWriteSrcError! Bookmark not defined.
	OK : BOOL;	TRUE if successful
Description	This function starts a divergence the end of the current branch. It automatically initiates the first branch (left side) so you can start instering items on it. The divergence will have to be ended by a convergence, even if its branches finish by a jump symbol (and thus do not actually converge). Divergences can be nested.	

Example



Functions paSfcBranch add a branch to the last open divergence

Syntax `OK := paSfcBranch (FID)`

Parameter `FID : DINT;` File identifier answered by
`paFileOpenWriteSrcError! Bookmark not defined.`

`OK : BOOL;` TRUE if successful

Description This function adds a new branch on the right of the divergence created by the last call to `paSfcDiv()` (**Error! Bookmark not defined.**) It is not needed to call this function for the first branch (on the left) as it is automatically initiated when the divergence is created.

Example

Functions paSfcConv Complete a divergence

Syntax `OK := paSfcConv (FID)`

Parameter `FID : DINT;` File identifier answered by
`paFileOpenWriteSrcError! Bookmark not defined.`

`OK : BOOL;` TRUE if successful

Description This function closes the divergence created by the last call to `paSfcDiv()` (**Error! Bookmark not defined.**) The divergence will have to be ended by a convergence, even if its branches finish by a jump symbol (and thus do not actually converge). Divergences can be nested.

Example



Functions `paSfcAddLv2String` add text string to level2 programming

Syntax `OK := paSfcAddLv2String (FID, KIND, TEXT)`

Parameter

<code>FID : DINT;</code>	File identifier answered by <code>paFileOpenWriteSrcError!</code> Bookmark not defined.
<code>KIND : STRING;</code>	Kind of level 2 Information
<code>TEXT : STRING;</code>	Text to be appended
<code>OK : BOOL;</code>	<i>TRUE</i> if successful

Description This function appends a string to the level 2 programming of the last created step or transition. It must be called immediately after the step or transition is created. The *KIND* argument specifies which piece of level 2 Information must be written. The following values are predefined:

Value	Description
<code>_LV2_NOTE</code>	Description text
<code>_LV2_COND</code>	Condition (for a transition)
<code>_LV2_ACTION</code>	Default actions of a step
<code>_LV2_P1</code>	P1 actions of a step
<code>_LV2_N</code>	N actions of a step
<code>_LV2_NOTE</code>	Description text
<code>_LV2_COND</code>	Condition (for a transition)
<code>_LV2_P0</code>	P0 actions of a step.

Generating level 2 programs in LD or FBD is not supported.

Example



Functions paSfcAddLv2Line add text line to level2 programming

Syntax `OK := paSfcAddLv2Line (FID, KIND, TEXT)`

Parameter

FID : DINT;	File identifier answered by paFileOpenWriteSrcError! Bookmark not defined.
KIND : STRING;	Kind of level 2 Information
TEXT : STRING;	Text to be appended
OK : BOOL;	<i>TRUE</i> if successful

Description This function appends a string plus end of lines characters ('\$R\$N') to the level 2 programming of the last created step or transition. It must be called immediately after the step or transition is created. The *KIND* argument specify which piece of level 2 Information must be written. The following values are predefined:

Value	Description
_LV2_NOTE	Description text
_LV2_COND	Condition (for a transition)
_LV2_ACTION	Default actions of a step
_LV2_P1	P1 actions of a step
_LV2_N	N actions of a step
_LV2_P0	P0 actions of a step

Generating level 2 programs in LD or FBD is not supported.

Example



Functions **paSfcAddLv2Eol** add end of line characters to level2 programming

Syntax **OK := paSfcAddLv2Eol (FID, KIND)**

Parameter FID : DINT; File identifier answered by
paFileOpenWriteSrcError! **Bookmark not defined.**

KIND : STRING; Kind of level 2 Information - see notes.

OK : BOOL; *TRUE* if successful.

Description This function appends end of lines characters ('\$R\$N') to the level 2 programming of the last created step or transition. It must be called immediately after the step or transition is created. The *KIND* argument specify which piece of level 2 Information must be written. The following values are predefined:

Value	Description
_LV2_NOTE	Description text
_LV2_COND	Condition (for a transition)
_LV2_ACTION	Default actions of a step
_LV2_P1	P1 actions of a step
_LV2_N	N actions of a step
_LV2_P0	P0 actions of a step

Generating level 2 programs in LD or FBD is not supported.

Example Below is an Example of script that generates a SFC file in the target project:

```
// make LD code
f := paFileOpenWriteProgramSrc ('ProgSFC');
if f <> 0 then

    // initial step
    paSfcIStep (f, 1);
    paSfcAddLv2String (f, _LV2_NOTE, 'Thats S1');
    paSfcAddLv2Line (f, _LV2_ACTION, 'VarXX (N);');

    // an 'OR' divergence
    paSfcDiv (f);

    paSfcTrans (f, 1);
    paSfcAddLv2String (f, _LV2_NOTE, 'Thats T1');
    paSfcAddLv2Line (f, _LV2_COND, 'condition');
```

```
// second branch
paSfcBranch (f);

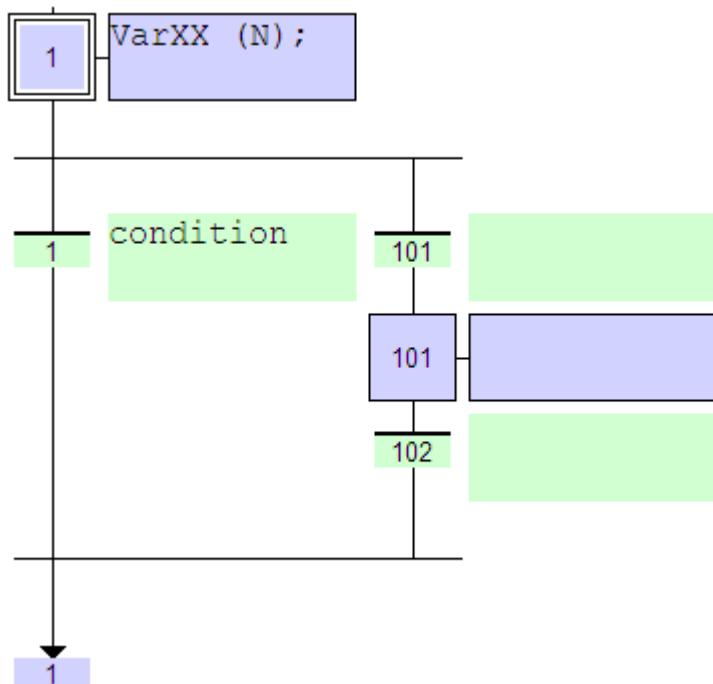
paSfcTrans (f, 101);
paSfcStep (f, 101);
paSfcTrans (f, 102);

// converge
paSfcCnv (f);

// jump to the initial step
paSfcJump (f, 1);

// close the file
paFileClose (f);
end_if;
```

Here is the SFC program generated by the script:





6.4. Network and Fieldbuses File Writing Services

Below are the main services for writing configuration files for networking or fieldbus I/O drivers. Files are open using specific functions. When complete, they must be closed by calling the paFileClose function.

Functions for MODBUS configuration:

paFileOpenWriteModbus	Open the MODBUS configuration for writing
paFModbusSlave	Configurate the MODBUS slave connection
paFModbusMaster	Create a MODBUS master connection
paFModbusRequestSlave	Add a MODBUS request on the slave connection
paFModbusRequestMaster	Add a MODBUS request on a master connection
paFModbusVariable	Add a MODBUS variable to the last declared request

Functions for AS-i configuration:

paFileOpenWriteASI	Open the AS-i configuration for writing
paFAsiApplyConfig	Set main flag
paFAsiMaster	Add a master device
paFAsiMasterDiag	Add a diagnostic variable on the last declared master
paFAsiSlave	Add a slave device
paFAsiSlaveDiag	Add a diagnostic variable on the last declared slave
paFAsiInput	Add an input variable on the last declared slave
paFAsiOutput	Add an output variable on the last declared slave

Functions for other fieldbus drivers:

paFileOpenWriteFieldbus	Open a feldbus configuration for writing
paFfbCreateMaster	Add a <i>master</i> node
paFfbCreateSlave	Add a <i>slave</i> node
paFfbCreateVar	Add a <i>variable</i> node
paFfb SetProperty	Set the the value of a property
paFfbSelectRoot	Select the root node
paFfbSelectMaster	Select a <i>master</i> node
paFfbSelectSlave	Select a <i>slave</i> node
paFfbSelectVar	Select a <i>variable</i> node

Functions for the binding configuration:

paFileOpenWriteBinding	Open the binding configuration for writing
paBindPublicVar	Add a public variable
paBindExternPort	Add an extern port
paBindCnxStatus	Add a connection status variable



paBindExternVar	Add an extern variable
paBindVarStatus	Add a status variable
paBindVarDateStamp	Add a date stamp variable
paBindVarTimeStamp	Add a time stamp variable



6.4.1. Network and Fieldbuses File Writing Functions

The following functions are the network and fieldbus file writing functions.

Functions **paFileOpenWriteModbus** Open the MODBUS configuration for writing

Syntax **FID := paFileOpenWriteModbus ()**

Parameter FID : DINT; File identifier or 0 if fail.

Description This function opens the MODBUS configuration file for writing. This function removes the contents of the configuration if it already exists. When written, the file must be closed by calling the paFileClose function.

Example

Functions **paFModbusSlave** Set the MODBUS slave configuration

Syntax **OK := paFModbusSlave (FID, SLAVENO)**

Parameter FID : DINT; File identifier returned by the paFileOpenWriteModbus function

SLAVENO : DINT; Default slave number for the MODBUS Slave server

OK : BOOL; TRUE if successful

Description This function defines the slave number used for the MODBUS slave server.

Example



Functions	paFModbusMaster	Add a MODBUS master port
Syntax	OK := paFModbusMaster (FID, OPENMODBUS, ADDRESS, PORT, RECONNECT)	
Parameter	FID : DINT;	File identifier returned by the paFileOpenWriteModbus function
	OPENMODBUS : BOOL;	TRUE = Open Modbus protocol / FALSE = RTU protocol
	ADDRESS : STRING;	Configuration of the port or IP address
	PORt : DINT;	IP port number in case of MODBUS on ETHERNET
	RECONNECT : BOOL;	Option: tries to reconnect after error.
	OK : BOOL;	TRUE if successful
Description	This function adds a new MODBUS master connection to the configuration.	

Example



Functions paFModbusRequestSlave Add a MODBUS slave request

Syntax `OK := paFModbusRequestSlave (FID, REQUEST, ADDRESS, NBITEM, NAME)`

Parameter	FID : DINT;	File identifier returned by the paFileOpenWriteModbus function
	REQUEST : STRING;	Type of request
	ADDRESS : DINT;	Base MODBUS address
	NBITEM : DINT;	Number of items (bits or words)
	NAME : STRING;	Optionnal description text
	OK : BOOL;	TRUE if successful

Description This function adds a new MODBUS slave request. You can define variables of the request just after calling this function.

 **Attention**

This function does not take care of address offsets defined by MODBUS. The first available address is always 0.

The following values are possible for the REQUEST parameter:

Value	Description
_MB_I_REG	Input registers
_MB_H_REG	Holding registers
_MB_I_BIT	Input bits
_MB_C_BIT	Coil bits

Example



Functions paFModbusRequestMaster Add a MODBUS master request

Syntax `OK := paFModbusRequestMaster (FID, MODE, REQUEST,
SLAVENO, ADDRESS, NBITEM, PERIOD, TIMEOUT, TRIALS,
NAME)`

Parameter		
FID : DINT;	File identifier returned by the paFileOpenWriteModbus function	
MODE : STRING;	Activation mode	
REQUEST : STRING;	Type of request	
SLAVENO : DINT;	Slave number of the device	
ADDRESS : DINT;	Base MODBUS address	
NBITEM : DINT;	Number of items (bits or words)	
PERIOD : TIME;	Activation period (for periodic requests)	
TIMEOUT : TIME;	Exchange timeout	
TRIALS : DINT;	Number of successive trials	
NAME : STRING;	Optionnal description text	
OK : BOOL;	TRUE if successful	

Description This function adds a new MODBUS master request. You can define variables of the request just after calling this function.

Attention

This function does not take care of address offsets defined by MODBUS. The first available address is always 0.

The following values are possible for the MODE parameter:

Value	Description
<code>_MB_PERIODIC</code>	Request activated periodically
<code>_MB_ONCALL</code>	Request activated by a variable
<code>_MB_ONCHANGE</code>	Request activated on a change of data (for "write" requests)



The following values are possible for the REQUEST parameter:

Value	Description
_MB_READ_I_REG	Read Input registers
_MB_READ_H_REG	Read Holding registers
_MB_WRITE_H_REG	Write Holding registers
_MB_READ_I_BIT	Read Input bits
_MB_READ_C_BIT	Read Coil bits
_MB_WRITE_C_BIT	Write Coil bits

Example



Functions	paFModbusVariable	Add a MODBUS variable
Syntax	OK := paFModbusVariable (FID, MODE, OFFSET, MASK, TWOWORDS, SYMBOL)	
Parameter	FID : DINT;	File identifier returned by the paFileOpenWriteModbus function
	MODE : STRING;	Exchange mode
	OFFSET : DINT;	Offset in the request
	MASK : DINT;	Mask for data exchange
	TWOWORDS : BOOL;	TRUE if data stored on 2 consecutive words
	SYMBOL : STRING;	Symbol of the variable
	OK : BOOL;	TRUE if successful
Description	This function connects a variable to the request defined by the last call to paFileModbusRequestSlave or paFileModbusRequestMaster functions.	
The following values are possible for the MODE parameter:		
Value	Description	
_MB_DATA	Data exchange	
_MB_STATUS	The variable receives the connection status	
_MB_COMMAND	The variable is used for activating the request	
_MB_TRIAL	The variable receives the current trial number	
Example		



Functions	paFileOpenWriteASi	Open the AS-i configuration for writing
Syntax	OK := paFModbusSlave (FID, SLAVENO)	
Parameter	FID : DINT;	File identifier or 0 if fail.
Description	This function opens the AS-i configuration file for writing. This function removes the contents of the configuration if it already exist. When written, the file must be closed by calling the paFileClose (Error! Bookmark not defined.) function.	

Example

Functions	paFAsiApplyConfig	Set AS-i configuration main flags
Syntax	OK := paFAsiApplyConfig (FID, APPLY)	
Parameter	FID : DINT;	File identifier returned by the paFileOpenWriteASi function
	APPLY : BOOL;	If TRUE, the AS-i configuration is applied when the runtime starts
	OK : BOOL;	TRUE if successful
Description	This function sets the main AS-i configuration flags for applying the AS-i configuration at runtime.	

Example

Functions	paFAsiMaster	Add an AS-i master
Syntax	OK := paFAsiMaster (FID, NAME, SETTINGS)	
Parameter	FID : DINT;	File identifier returned by the paFileOpenWriteASi function
	NAME : STRING;	Description text
	SETTINGS : STRING;	AS-i controller configuration string
	OK : BOOL;	TRUE if successful
Description	This function adds a master device to the AS-i configuration. AS-i slaves and diagnostic variables for this master must be added just after calling this function.	

Example

**Functions** paFAsiMasterDiag

Add an AS-i master diagnostic variable

Syntax **OK := paFAsiMasterDiag (FID, NAME, DIAG)**

Parameter	FID : DINT;	File identifier returned by the paFileOpenWriteASi function
	NAME : STRING;	Symbol of the variable
	DIAG : DINT;	Desired diagnostic Information
	OK : BOOL;	TRUE if successful

Description This function adds a diagnostic variable to the last declared AS-i master.

The possible values are available for the DIAG parameter:

Value	Description
_ASI_M_CONFOK	Configuration OK
_ASI_M_SLV0	Slave detected at address 0
_ASI_M_AUTOACT	Auto-addressing is active
_ASI_M_AUTOAV	Auto-addressing is available
_ASI_M_CNFACT	Configuration is active
_ASI_M_NORMAL	Normal mode
_ASI_M_PWFAIL	Power fail detected
_ASI_M_OFFLINE	Off Line mode
_ASI_M_PERIPHOK	Periphery OK

Example



Functions	paFAsiSlave	Add an AS-i slave
Syntax <code>OK := paFAsiSlave (FID, NAME, ADDRESS, PROFILE)</code>		
Parameter	<code>FID : DINT;</code>	File identifier returned by the paFileOpenWriteASi function
	<code>NAME : STRING;</code>	Description text
	<code>ADDRESS : DINT;</code>	AS-i slave address
	<code>PROFILE : STRING;</code>	Slave profile written in AS-i convention: 'H.H.H.H' (hexadecimal digits)
	<code>OK : BOOL;</code>	TRUE if successful
Description	This function adds a slave device under the last declared AS-i master. The AS-i master variables must be added for this slave just after calling this function.	
Example		

**Functions paFAsiSlaveDiag**

Add an AS-i slave diagnostic variable

Syntax **OK := paFAsiSlaveDiag (FID, NAME, DIAG)**

Parameter	FID : DINT;	File identifier returned by the paFileOpenWriteASi function
	NAME : STRING;	Symbol of the variable
	DIAG : DINT;	Desired diagnostic Information
	OK : BOOL;	TRUE if successful

Description This function adds a diagnostic variable to the last declared AS-i slave.

The possible values are available for the DIAG parameter:

Value	Description
_ASI_S_ACT	Slave active
_ASI_S_DET	Slave detected
_ASI_S_PRJ	Slave projected
_ASI_S_COR	Slave corrupted
_ASI_S_FLT	Periphery fault

Example



Functions	paFAsiInput	Add an AS-i input variable
	Syntax	OK := paFAsiInput (FID, NAME, OFFSET)
	Parameter	FID : DINT; File identifier returned by the paFileOpenWriteASI function
		FID : DINT; File identifier returned by the paFileOpenWriteASI function
		NAME : STRING; Symbol of the variable
		OFFSET : DINT; Input number (1 to 4) or 0 for all inputs in a byte
		OK : BOOL; TRUE if successful
	Description	This function adds an input variable to the last declared AS-i slave.

Example

Functions	paFAsiOutput	Add an AS-i output variable
	Syntax	OK := paFAsiOutput (FID, NAME, OFFSET)
	Parameter	FID : DINT; File identifier returned by the paFileOpenWriteASI function
		NAME : STRING; Symbol of the variable
		OFFSET : DINT; Output number (1 to 4) or 0 for all inputs in a byte
		OK : BOOL; TRUE if successful
	Description	This function adds an output variable to the last declared AS-i slave.

Example



Functions **paFileOpenWriteFieldbus** Open the AS-i configuration for writing**Syntax** **FID := paFileOpenWriteFieldbus (CONFIG)****Parameter** CONFIG : STRING Type of fieldbus

FID : DINT; File identifier or 0 if fail

Description This function opens the selected fieldbus configuration file for writing. This function removes the contents of the configuration if it already exist. When written, the file must be closed by calling the paFileClose (**Error! Bookmark not defined.**) function.

The *CONFIG* string is the prefix of the wizard DLL used for configuration. Available DLLs are stored in the \IOD folder where the Workbench is installed. You can also obtain the list of available configurations from the **Prompt** tab in the output window. Simply enter the following ?FB command.

Example **>?FB****Fieldbus configuration wizards:****K5BusAppIO: ApplicomIO configuration****K5BusCAN: CAN bus****K5BusHMS: Anybus configuration****K5BusINTERBUS: Interbus-S****K5BusIOTools: Brodersen IOTOOLS****K5BusProfDP: Hilscher CIF Profibus****K5BusProfNetIO: PROFINET IO****K5BusShm: Win32 Shared memory****K5BusSoftNet: SoftNet ProfibusDP****K5BusSycon: Hilscher SYCON configuration****K5BusXFLOW: Xflow****K5ThinkIO: ThinkIO/IOSystem 758****K5ZenOnRT: Run**



Functions	paFfbCreateMaster	Add a master node
Syntax OK := paFfbCreateMaster (FID)		
Parameter	FID : DINT;	File identifier returned by the paFileOpenWriteFieldbus function
	OK : BOOL;	TRUE if successful
Description This function adds a <i>master</i> node to the configuration.		
Example		

Functions	paFfbCreateSlave	Add a slave node
Syntax OK := paFfbCreateSlave (FID)		
Parameter	FID : DINT;	File identifier returned by the paFileOpenWriteFieldbus function
	OK : BOOL;	TRUE if successful
Description This function adds a <i>slave</i> node to the configuration. The node is added under the last created or selected "master" node.		
Example		

Functions	paFfbCreateVar	Add a <i>variable</i> node
Syntax OK := paFfbCreateVar (FID)		
Parameter	FID : DINT;	File identifier returned by the paFileOpenWriteFieldbus function
	OK : BOOL;	TRUE if successful
Description This function adds a <i>variable</i> node to the configuration. The node is added under the last created or selected <i>slave</i> node.		
Example		



Functions	paFfbSetProperty	Set the value of a property
Syntax	OK := paFfbSetProperty (FID, PROP, VALUE)	
Parameter	FID : DINT;	File identifier returned by the paFileOpenWriteFieldbus function
	PROP : STRING	Internal name of the property
	VALUE : STRING	Value of the property in neutral format
	OK : BOOL;	TRUE if successful
Description	This function adds a <i>variable</i> node to the configuration. The node is added under the last created or selected <i>slave</i> node. The name of the property must be entered in "neutral" form, independently from the language selected for the Workbench. Also, for a check box or a drop-list property, you need to enter the value in neutral form, and not as displayed in the fieldbus configuration tool. To know the relationship in between neutral and displayed names and values, go to the Prompt tab in the output window and hit the ?FB Command followed by the name of the configuration. This command lists all properties for each level of the selected configuration tree, giving neutral names and readable names, and lists all possible neutral values for Boolean and enumerated properties.	
Example	<code>//Example of syntax: >?FB K5BusCan</code>	



Functions	paFfbSelectRoot	Selects the root node
Syntax <code>OK := paFfbSelectRoot (FID)</code>		
Parameter	FID : DINT;	File identifier returned by the paFileOpenWriteFieldbus function
	OK : BOOL;	TRUE if successful
Description This function selects the root node of the configuration tree. The root node is automatically selected when the file is open. Selecting a node is required for setting its properties or adding child nodes. It is not work to select a node immediately after its creation.		

Example

Functions	paFfbSelectMaster	Selects a <i>master</i> node
Syntax <code>OK := paFfbSelectMaster (FID, MASTER)</code>		
Parameter	FID : DINT;	File identifier returned by the paFileOpenWriteFieldbus function
	MASTER : DINT	1-based index in the list of master nodes
	OK : BOOL;	TRUE if successful
Description This function selects a <i>master</i> in the configuration tree. Selecting a node is required for setting its properties or adding child nodes. It is not work to select a node immediately after its creation.		

Example



Functions	paFfbSelectSlave	Selects a slave node
Syntax	OK := paFfbSelectSlave (FID, MASTER, SLAVE)	
Parameter	FID : DINT;	File identifier returned by the paFileOpenWriteFieldbus function
	MASTER : DINT	1-based index of the parent master node
	SLAVE : DINT	1-based index in the list of slave nodes for the selected parent master
	OK : BOOL;	<i>TRUE</i> if successful
Description	This function selects a slave in the configuration tree. Selecting a node is required for setting its properties or adding child nodes. It is not work to select a node immediately after its creation.	

Example

Functions	paFfbSelectVar	Selects a Variable node
Syntax	OK := paFfbSelectVar (FID, MASTER, SLAVE, VARIABLE)	
Parameter	FID : DINT;	File identifier returned by the paFileOpenWriteFieldbus function
	MASTER : DINT	1-based index of the parent master node
	SLAVE : DINT	1-based index of the parent slave node
	VARIABLE : DINT	1-based index in the list of variable nodes for the selected parent slave
	OK : BOOL;	<i>TRUE</i> if successful
Description	This function selects a <i>variable</i> in the configuration tree. Selecting a node is required for setting its properties or adding child nodes. It is not work to select a node immediately after its creation.	

Example



Functions paFfbSelectVar

Selects a Variable node

Syntax `OK := paFfbSelectVar (FID, MASTER, SLAVE, VARIABLE)`

Parameter	FID : DINT;	File identifier returned by the paFileOpenWriteFieldbus function
	MASTER : DINT	1-based index of the parent master node
	SLAVE : DINT	1-based index of the parent slave node
	VARIABLE : DINT	1-based index in the list of variable nodes for the selected parent slave
	OK : BOOL;	TRUE if successful

Description This function selects a *variable* in the configuration tree. Selecting a node is required for setting its properties or adding child nodes. It is not work to select a node immediately after its creation.**Example**

Functions paFileOpenWriteBinding

Open the binding configuration for writing

Syntax `FID := paFileOpenWriteBinding ()`

Parameter	FID : DINT;	File identifier or 0 if fail
------------------	-------------	------------------------------

Description This function opens the binding configuration file for writing. This function removes the contents of the configuration if it already exist. When written, the file must be closed by calling the paFileClose function.**Example**



Functions	paBindPublicVar	Add a published variable to the binding configuration
Syntax FID := paBindPublicVar (FID, ID, NAME, HYSTP, HYSTN)		
Parameter	FID : DINT;	File identifier returned by the paFileOpenWriteBinding function
	ID : DINT;	ID of the binding link
	NAME : STRING;	Symbol of the variable
	HYSTP : STRING;	Positive hysteresis in IEC syntax (or empty string)
	HYSTN : STRING;	Negative hysteresis in IEC syntax (or empty string)
	OK : BOOL;	TRUE if successful
Description	This function adds a public variable in the binding configuration.	

Example

Functions	paBindExternPort	Add an external port to the binding configuration
Syntax FID := paBindExternPort (FID, ADDR, PORT, NAME)		
Parameter	FID : DINT;	File identifier returned by the paFileOpenWriteBinding function
	ADDR : STRING;	IP address
	PORT : DINT;	IP port number
	NAME : STRING	Optional description text
	OK : BOOL;	TRUE if successful
Description	This function adds a public variable in the binding configuration. Immediately after calling this function, you can use other functions to declare external bound variables.	

Example



Functions **paBindCnxStatus** Binds a variable to the connection status of the last declared port

Syntax **FID := paBindCnxStatus (FID, NAME)**

Parameter FID : DINT; File identifier returned by the paFileOpenWriteBinding function

NAME : STRING; Symbol of the bound variable

OK : BOOL; TRUE if successful

Description This function binds a variable for getting the connection status. The variable is bound to external port declared by the last call to the paBindExternPort function.

Example

Functions **paBindExternVar** Binds a variable to the last declared port

Syntax **FID := paBindExternVar (FID, ID, NAME)**

Parameter FID : DINT; File identifier returned by the paFileOpenWriteBinding function

ID : DINT; ID of the binding link

NAME : STRING; Symbol of the bound variable

OK : BOOL; TRUE if successful

Description This function binds a variable for data exchange. The variable is bound to external port declared by the last call to the paBindExternPort function.

Example



Functions **paBindVarStatus** Binds a status variable to the last declared port

Syntax **FID := paBindVarStatus (FID, ID, NAME)**

Parameter FID : DINT; File identifier returned by the paFileOpenWriteBinding function
ID : DINT; ID of the binding link
NAME : STRING; Symbol of the bound variable
OK : BOOL; TRUE if successful

Description This function binds a variable for getting the status of a binding link. The variable is bound to external port declared by the last call to the paBindExternPort function.

Example

Functions **paBindVarDateStamp** Binds a date stamp variable to the last declared port

Syntax **FID := paBindVarDateStamp (FID, ID, NAME)**

Parameter FID : DINT; File identifier returned by the paFileOpenWriteBinding function
ID : DINT; ID of the binding link
NAME : STRING; Symbol of the bound variable
OK : BOOL; TRUE if successful

Description This function binds a variable for getting the date stamp of a binding link. The variable is bound to external port declared by the last call to the paBindExternPort function.

Example



Functions **paFbindVarTimeStamp** Binds a time stamp variable to the last declared port

Syntax **FID := paFbindVarTimeStamp (FID, ID, NAME)**

Parameter FID : DINT; File identifier returned by the
paFileOpenWriteBinding function

ID : DINT; ID of the binding link

NAME : STRING; Symbol of the bound variable

OK : BOOL; TRUE if successful

Description This function binds a variable for getting the time stamp of a
binding link. The variable is bound to external port declared
by the last call to the paBindExternPort function.

Example



6.5. Watch Window Documents File Writing Services

Below are the main services for writing watch window document files. Files are open using specific functions. When complete, the paFileClose function must be called to close the file.

Generating spy lists:

paFileOpenWriteSpyList Open a spy list document for writing

paFSpyAddVariable Add a variable to a spy list

Generating recipes:

paFileOpenWriteRecipe Open a recipe document for writing

paFrcpAddColumn Define a column in the recipe

paFrcpAddVariable Add a variable to a recipe

paFrcpAddValue Add a value to a recipe

Generating graphic documents:

Not available



6.5.1. Watch Window Documents File Writing Functions

The following functions are the watch window documents file writing functions.

Functions	paFileOpenWriteSpyList	Open a spy list document for writing
	Syntax	FID := paFileOpenWriteSpyList (NAME)
	Parameter	NAME : STRING; Name of the spy list (no suffix!)
		FID : DINT; File identifier or 0 if fail
	Description	This function opens a spy list document file for writing. It removes the contents of the spy list if it already exists. When written, the file must be closed by calling the paFileClose function.

Example

Functions	paFSpyAddVariable	Add a variable to a spy list
	Syntax	OK := paFSpyAddVariable (FID, NAME)
	Parameter	FID : DINT; File identifier returned by the paFileOpenSpyList function
		NAME : STRING; Symbol of the variable
		OK : BOOL; TRUE if successful
	Description	This function adds a variable to an open spy list document.

Example

Functions	paFileOpenWriteRecipe	Open a recipe document for writing
	Syntax	FID := paFileOpenRecipe (NAME)
	Parameter	NAME : STRING; Name of the recipe (no suffix!)
		FID : DINT; File identifier or 0 if fail
	Description	This function opens a recipe document file for writing, and removes the contents of the recipe if it already exist. When written, the file must be closed by calling the paFileClose function.

Example



Functions	paFrcpAddColumn	Add a column to a recipe
Syntax OK := paFrcpAddColumn (FID, COL, NAME)		
Parameter	FID : DINT;	File identifier returned by the paFileOpenRecipe function
	COL : DINT;	Index of the column (0 based)
	NAME : STRING;	Name of the column
	OK : BOOL;	TRUE if successful
Description	This function adds a column to a recipe. The index of the first column is 0. The Column must be declared from the left to the right and defined before adding values to the recipe.	

Example

Functions	paFrcpAddVariable	Add a variable to a recipe
Syntax OK := paFrcpAddVariable (FID, NAME)		
Parameter	FID : DINT;	File identifier returned by the paFileOpenRecipe function
	NAME : STRING;	Symbol of the variable
	OK : BOOL;	TRUE if successful
Description	This function adds a variable (a line) to an open recipe document. Variables must be defined before adding values to the recipe.	

Example



Functions paFrcpAddValue

Add a value to a recipe

Syntax `OK := paFrcpAddValue (FID, NAME, COL, VALUE)`**Parameter** `FID : DINT;` File identifier returned by the `paFileOpenRecipe` function`NAME : STRING;` Symbol of the variable`COL : DINT;` Index of the column (0 based)`NAME : STRING;` Value in IEC syntax`OK : BOOL;` TRUE if successful**Description** This function adds a value (one cell) to a recipe. The corresponding variable (line) and column must be defined before calling this function.

Example



6.6. Resource Documents File Writing Services

Below are the main services for writing resources document files. Files are open using specific functions. When complete, the paFileClose function must be called to close the file.

Generating string tables:

paFileOpenWriteStringTable	Open a string table document for writing
paFstbAddColumn	Define a column in the string table
paFstbAddString	Add a string to a table

Generating analog signal:

paFileOpenWriteSignal	Open a signal document for writing
paFSignalAddColumn	Define a column in the signal resource
paFSignalAddPoint	Add a point to a signal resource



6.6.1. Resource Documents File Writing Functions

The following functions are the resource documents file writing functions.

Functions **paFileOpenWriteStringTable** Open a "string table" resource document for writing

Syntax **FID := paFileOpenStringTable (NAME)**

Parameter NAME : STRING; Name of the string table (no suffix!)

FID : DINT; File identifier or 0 if fail

Description This function opens a "string table" resource document file for writing, and removes the contents of the stringtable if it already exist. When written, the file must be closed by calling the paFileClose function.

Example

Functions **paFstbAddColumn** Add a column to a string table resource

Syntax **OK := paFstbAddColumn (FID, NAME)**

Parameter FID : DINT; File identifier returned by the paFileOpenStringTable function

COL : DINT; Index of the column (0 based)

NAME : STRING; Name of the column

OK : BOOL; TRUE if successful

Description This function adds a column to a "string table" resource. The index of the first column is 0. Column must be declared from the left to the right. Columns must be defined before adding strings to the table.

Example



Functions paFstbAddString

Add a column to a string table resource

Syntax `OK := paFstbAddString (FID, NAME, COL, VALUE)`

Parameter `FID : DINT;` File identifier returned by the `paFileOpenStringTable` function
`NAME : STRING;` Identifier of the string in the table
`COL : DINT;` Index of the column (0 based)
`VALUE : STRING;` Contents of the string
`OK : BOOL;` TRUE if successful

Description This function adds a string to a "string table" resource. The corresponding column must be defined before calling this function.

Example

Functions paFileOpenWriteSignal

Open a "signal" resource document for writing

Syntax `FID := paFileOpenSignal (NAME)`

Parameter `NAME : STRING;` Name of the signal (no suffix!)
`FID : DINT;` File identifier or 0 if fail

Description This function opens an "analog signal" resource document file for writing, and removes the contents of the resource if it already exist. When written, the file must be closed by calling the `paFileClose` function.

Example



Functions **paFSignalAddColumn** Adds a "signal column for writing

Syntax `OK := paFSignalAddColumn (FID, NAME)`

Parameter `FID : DINT;` File identifier returned by the `paFileOpenSignal` function
`COL : DINT;` Index of the column (0 based)
`NAME : STRING;` Name of the column
`OK : BOOL;` TRUE if successful

Description This function adds a column to an "analog signal" resource. The index of the first column is 0. The Column must be declared from the left to the right and must be defined before adding points are added to the signal.

Example

Functions **paFSignalAddPoint** Add a point to a signal resource

Syntax `OK := paFSignalAddPoint (FID, TM, COL, VALUE)`

Parameter `FID : DINT;` File identifier returned by the `paFileOpenSignal` function
`TM : TIME;` X coordinate (time) of the point
`COL : DINT;` Index of the column (0 based)
`VALUE : REAL;` Y coordinate of the point
`OK : BOOL;` TRUE if successful

Description This function adds a point to an "analog signal" resource. The corresponding column must be defined before calling this function.

Example



6.7. Templates

6.7.1. Variable Keywords For Copying Templates

Scripts can contain template documents to be copied in the target project. When copying documents, some special keywords may be variables defined from the script and replaced by the value given by the script.

Use the following syntax in your templates for specifying variable keywords:

`$(keyname)`

and use the following functions in your script to set values for keywords:

`paSetKey` Define the value of a variable keyword

`paRemoveKeys` Reset values of all defined variable keywords

6.7.2. Variable Keyword Functions

The following functions are the variable keyword functions.

Functions	<code>paSetKey</code>	Define a variable keyword for copying templates
	Syntax	<code>OK := paSetKey (NAME, VALUE)</code>
	Parameter	NAME : STRING; Name of the variable keyword
		VALUE : STRING; Value of the variable keyword
		OK TRUE if successful
	Description	This function defines the value of a defined keyword to be replaced when copying a template.
	Example	Template ST program: <pre>if ALARM_\$(ALM) then bMainAlarm := TRUE; end_if;</pre> Script instructions: <pre>paSetKey ('ALM', 'Engine2') paTplCopySource ('TargetProgram', 'TemplateName');</pre> Generated ST program: <pre>if ALARM_Engine2 then bMainAlarm := TRUE; end_if;</pre>



Functions	paRemoveKeys	Remove all values defined for template variable keywords
	Syntax	OK := paRemoveKeys ()
	Parameter	OK TRUE if successful
	Description	This function removes all values defined for variable keywords used for copying templates.

Example

6.7.3. Templates of programs

Scripts can contain template documents to be copied in the target project. You can include in your script project templates of POU (programs, sub-programs or UDFBs) to be copied in the target project.

Use the following functions in your script to copy a template of POU:

paTplCopySource	Copy the source code
paTplCopyVars	Copy local variables
paTplCopyDefinitions	Copy local definitions
paTplCopyAll	Copy all components of a POU template

You also can define some variable keywords to be replaced when copying templates of POU.

6.7.4. Template Variable Keyword Functions

The following functions are the template variable keyword functions.

Functions	paTplCopySource	Copy the source code of a POU template
	Syntax	OK := paTplCopySource (DSTPROG, TEMPLATE)
	Parameter	DSTPROG : STRING; Name of the destination program in the target project (must exist)
		TEMPLATE : STRING; Name of the template program in the automation script project
		OK TRUE if successful
	Description	This function copies the source code of the specified template from the automation script project to the destination project. The destination program must be declared in the target project.

Example



Functions	paTplCopyVars	Copy local variables of a POU template
	Syntax	OK := paTplCopyVars (DSTPROG, TEMPLATE)
	Parameter	DSTPROG : STRING; Name of the destination program in the target project (must exist)
		TEMPLATE : STRING; Name of the template program in the automation script project
		OK <i>TRUE</i> if successful
	Description	This function copies all local variables of the specified template from the automation script project to the destination project. The destination program must be declared in the target project.

Example

Functions	paTplCopyDefinitions	Copy local definitions of a POU template
	Syntax	OK := paTplCopyDefinitions (DSTPROG, TEMPLATE)
	Parameter	DSTPROG : STRING; Name of the destination program in the target project (must exist)
		TEMPLATE : STRING; Name of the template program in the automation script project
		OK <i>TRUE</i> if successful
	Description	This function copies all local definitions of the specified template from the automation script project to the destination project. The destination program must be declared in the target project.

Example



Functions **paTplCopyAll**

Copy a POU template

Syntax **OK := paTplCopyAll (DSTPROG, TEMPLATE)**

Parameter DSTPROG : STRING; Name of the destination program in the target project (must exist)

TEMPLATE : STRING; Name of the template program in the automation script project

OK *TRUE* if successful

Description This function copies the specified template from the automation script project to the destination project. The copy includes source code, local variables and local definitions. The destination program must be declared in the target project.

Example



6.8. Script parameters

The project automation system enables your script to prompt the user to enter parameters. These parameters are then used (as variables) by your script.

Parameters must be variables of your script project declared as *global* variables. Create a list of variables in your script project that groups all "parameter" variables, and within your script call the paEditParameterList function to open a dialog box where the user provides values for each parameter. Then "parameter" variables can then be used in your script.

A parameter must be a variable declared with a single data type (no array, no structure). All basic data types are supported. When the parameter input dialog box is open, current values of variables are shown as default value.

For a string parameter, it is possible to define a drop-list choice. For that you must set possible choices in the string value before editing parameter, separated by '|' characters. e.g.: 'Choice1|Choice2'.

The following are script related functions:

paSetParameterDesc Set the description text to be shown with a parameter.

paEditParameterList Prompt the user for entering parameters.

6.8.1. Script Related Functions

Functions	paSetParameterDesc	Sets the description text to be shown with a parameter
		This function defines the description text to be shown together with the specified parameter when entered.
	Syntax	<code>OK:= paSetParameterDesc (NAME, DESC)</code>
	Parameter	NAME: STRING; Name of the parameter, such as declared in your script project
		DESC: STRING; Description text to be show to the user when entering parameters.
		OK: BOOL TRUE if successful.
	Description	This function copies the specified template from the automation script project to the destination project. The copy includes source code, local variables and local definitions. The destination program must be declared in the target project.

Example



Functions **paEditParameterList**

Prompt the user to enter parameter values

This function opens a dialog box where the user enters values for script parameters

Syntax **OK := paEditParameterList (LISTID, TITLE)**

Parameter LISTID: DINT; D of the list of parameters (use VLID function)

TITLE: STRING; Title of the window where the user enters the parameter

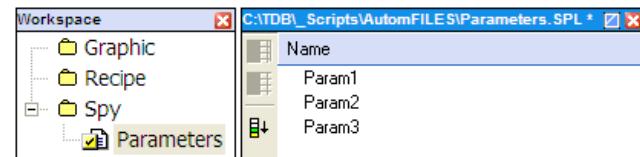
OK: BOOL TRUE if the user pressed the OK button

Example This Examples shows the different steps for adding parameters to a script:

1. Declare parameters as global variables of your script project

Main		
Name	Type	Init value
Global variables		
Param1	BOOL	TRUE
Param2	DINT	123
Param3	STRING(255)	'YES NO MAYBE'

2. Create a list of variables with parameters



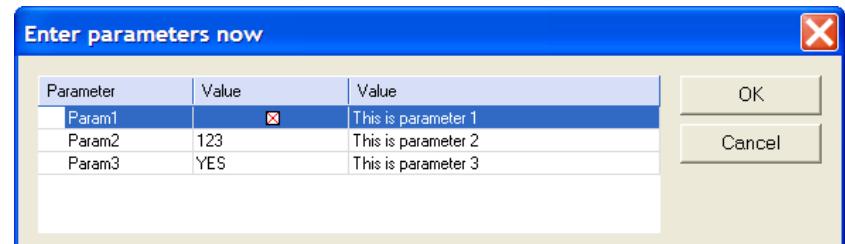
ST Script example

```
// set description text for parameters
paSetParameterDesc ('Param1', 'This is parameter 1');
paSetParameterDesc ('Param2', 'This is parameter 2');
paSetParameterDesc ('Param3', 'This is parameter 3');

// prompt the user for parameters

paEditParameterList (VLID ('Parameters'), 'Enter
parameters now');
```

When the script is run, the following dialog box is open



The *Param1* parameter is a check box, because it is declared as BOOL. The *Param3* is shown as a drop list because the value of *Param3* (its initial value in that case) is a list of words separated by '|' characters.



6.9. Miscellaneous

Below are some other functions you can use in automation scripts:

paGetLanguage Return the selected language for the Workbench

paTRACE... Output a report message

6.9.1.1. **paGetLanguage**

paGetLanguage Get current selected language

Syntax `LGE := paGetLanguage ()`

Parameter `LGE : DINT;` Language identifier (see notes)

Description This function returns the language currently selected for the Workbench.

Possible returned values are:

`_WB_USA` English

`_WB_GER` German

`_WB_FRA` French

`_WB_ITA` Italian

`_WB_SPA` Spanish

`_MW_KOR` Korean

6.9.1.2. **paTRACEEx**

paTRACE... Output a report message

Syntax `paTRACE0 (TEXT)`

`paTRACE1 (TEXT, ARG1)`

`paTRACE2 (TEXT, ARG1, ARG2)`

`paTRACE3 (TEXT, ARG1, ARG2, ARG3)`

`paTRACE4 (TEXT, ARG1, ARG2, ARG3, ARG4)`

Parameter `TEXT : STRING;` Text to be output

`ARG1..4 : DINT;` Arguments

Description The message specified in the TEXT argument is output to the report window when the automation script is run. The message text may include up to 4 integer arguments, specified in the TEXT string with special character sequences:

`%ld` signed value in decimal.

`%lu` unsigned value in decimal.

`%lx` value in hexadecimal.



Chapter 7: Workbench COM Interface

The Workbench offers a COM interface (K5COM.dll) which sets up on the program automation interface. As the referring object model is developed continuously this description could not be up to date.

The COM interfac can be use for e.g.:

- declaring variables via a VBA wizard.
- browsing programs or the variable lists.
- automatic programming....

This page describes the object model only. The remaing information can be investigated in the object library of the used programming environments (e.g. VBA). Here you find the object model represented as a tree:

- StratonCom
- Project
 - GlobalVariables
 - Variables
 - Variable
 - VarProps
 - VarProperty
 - RetainVariables
 - Variables
 - Variable
 - VarProps
 - VarProperty
 - Programs
 - Variables
 - Variable
 - VarProps
 - VarProperty
 - ProgramContent
 - ProgramObjects
 - GraphicalObject
 - ProgramLines
 - LineObject

You can use all program automation commands through the shell of the used programming language (e.g. VBA).

7.1.1. day_time_local

Function Reads the current MS Windows system time or the current date

Inputs IN :DINT Specifies the desired Information (see below).

Outputs Q : STRING Desired Information formatted on a string.

Remarks Possible values of IN input

- 1 current time - format: 'HH:MM:SS'
2 day of the week
0 (default) current date - format: 'YYYY/MM/DD'

Designed for COPA-DATA's Runtime for Windows XP and CE only.



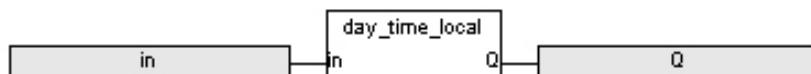
Must not be used for COPA-DATA's RTK (real-time)!

In LD language, the input rung (EN) enables the selection. The output rung keeps the same state as the input rung.

ST Language

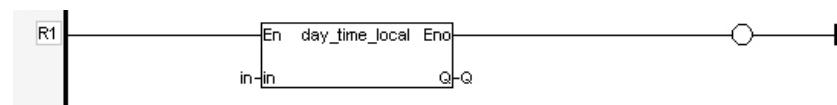
Q := day time local (IN);

FBD Language



LD Language

The Function is performed when the EN input is *TRUE*.



IL Language

Op1: LD IN day_time_local ST Q



Chapter 8: PID Functions by JS Automation

Function	Explanation	Call	Return
JS_DeadTime	analog delay	INPUT: signal input N: number of samples (200 max) DeadTime: delay (seconds)	OUT: output signal.
 Info			Allows to put a delay on an analog signal.
 Attention			The dead time divided by the number of samples must be very greater than cycle time. If N = 0 the function understands 1 If N > 200 the function understands 200
JS_LeadLag	signal lead / lag	Input : input signal. Lead : lead value. Lag : lag value. Ts : sampling period.	Out : output signal.
Qn = Qn-1 + Ts/Ti.(Mn-1 – Qn-1) + Td/Ti.(Mn - Mn-1) Qn : Output at t Qn-1 : Output at t-1 Mn : Input at t Mn-1 : Input at t-1 Ts : Sampling period Ti : Lag Td : Lead	JS_PID - PID loop setpoint balance	LSL : Loop Scale Lo LSH : Loop Scale Hi Auto : automatic or manual mode Pv : Process output value Sp : Set point value Ramp : Setpoint Ramp (Unit per Minute) Balance : Auto/Manu Setpoint balancing Action : Output action Direct or Reverse Mixt : Interactive or Non-Interactive PID Deriv : Derivative action Feedback : external PID feedback for output manipulations X0 : Adjustment value: In manual mode, output pid regulator equal to X0 You must connect obligatory a variable (not a constant and no computation on this variable) Kp : Proportionality constant Ti : Integral time constant in minute Td : derivative time constant in minute Ts : Sampling period Xmin : Minimum limit on output command value	SPcur: Current Setpoint Xout : Command



Function	Explanation	Call	Return
		Xmax : Maximum limit on output command value	
 Info			
If Lag = 0 the function is not executed. Sampling period must be larger than the cycle time.			
Automatic mode must be set to false at init. Balance = 0 Non-balancing Setpoint ; =1 Balancing Xout is limited inside specified Xmin ,Xmax range. Xmax should be greater than Xmin. The integral term is held when Xout reaches the limits. The Ts parameter should be greater (>>) than the kernel cycle time. Action : <ul style="list-style-type: none">• Direct (0) Output increase if PV-SP positive and decrease if PV-SP negative.• Reverse (1) Output decrease if PV-SP positive and increase if PV-SP negative. Mixt : <ul style="list-style-type: none">• Interactive PID (0) : The I and D parameters are multiplied by KP.• Non-Interactive (1) : The P,I,D parameters are independant. Feedback : <ul style="list-style-type: none">• The range must be 0-100 .• Feedback=0 : internal feedback.			
JS_Ramp	Limit variation speed	INPUT : input signal Rampe : maximum variation speed (Unit/mn) Cycle : application cycle time	OUT : output signal
 Info			
The variation of speed is expressed as units per minute The Cycle input must be the application cycle time Use GetSysInfo (_SYSINFO_CYCLETIME_MS)			

Chapter 9: IDE Library Functions

9.1.1. ROLb / ROL_SINT / ROL_USINT / ROL_BYTE

Function	Rotate bits of a register to the left
Inputs	IN : SINT 8 bit register NBR : SINT Number of rotations (each rotation is 1 bit)
Outputs	Q : SINT Rotated register
Diagram	
Remarks	In LD language, the input rung (EN) enables the operation, and the output rung keeps the state of the input rung. In IL language, the first input must be loaded before the function call. The second input is the operand of the function.
ST Language	<code>Q := ROLb (IN, NBR);</code>
FBD Language	
LD Language	The rotation is executed only if EN is <i>TRUE</i> . ENO has the same value as EN.
IL Language	<code>Op1: LD IN ROLb NBR ST Q</code>

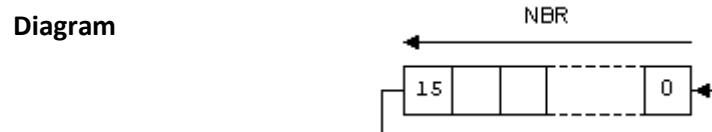
9.1.2. ROLw / ROL_INT / ROL_UINT / ROL_WORD

Function Rotate bits of a register to the left

Inputs IN : INT 16 bit register.

NBR : INT Number of rotations (each rotation is 1 bit).

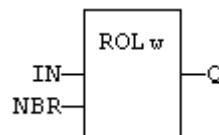
Outputs Q : INT Rotated register.



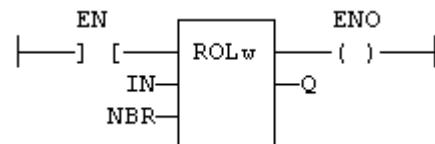
Remarks In LD language, the input rung (EN) enables the operation, and the output rung keeps the state of the input rung. In IL language, the first input must be loaded before the function call. The second input is the operand of the function.

ST Language `Q := ROLw (IN, NBR);`

FBD Language



LD Language The rotation is executed only if EN is *TRUE*. ENO has the same value as EN.



IL Language

`Op1: LD IN ROLw NBR ST Q`



9.1.3. ApplyRecipeColumn RTK_IsBugCheck

Function

Reads the Blue screen of Death value on MS Windows XP

Inputs

Outputs

Q : BOOL *TRUE* if a Blue screen of Death occurred.

Remarks

Designed for COPA-DATA's RTK Real-time Runtime for Windows XP only



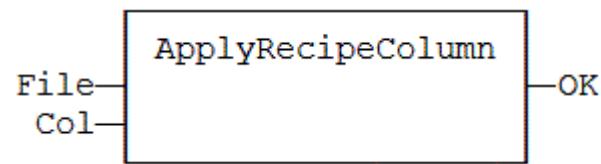
After a Blue screen of Death of MS Windows XP Windows specific functions in the PLC code can not be executed any more. These are: File operations, TCP functions, Serial interface functions, non real time fieldbus drivers, System clock functions, RETAIN variables, writing log messages

In LD language, the input rung (EN) enables the selection. The output rung is Q the output signal

ST Language

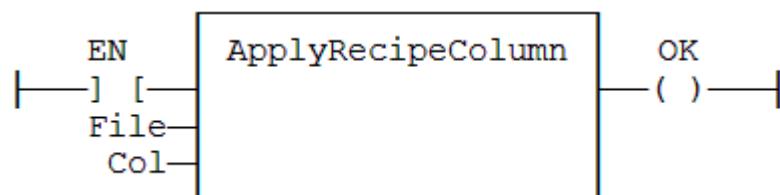
`Q := RTK_IsBugCheck();`

FBD Language



LD Language

The Function is performed when the EN input is *TRUE*



IL Language

Not available



9.1.4. Terminate

Function	Terminates the Runtime	
Inputs	I : BOOL	Input to terminate (close) the Runtime
	NBR : INT	Number of rotations (each rotation is 1 bit).
Outputs	Q : BOOL TRUE if successful	
Remarks	This function causes a termination of the Runtime	
ST Language	Q := Terminate(I);	
FBD Language	This function is available in FBD language	
LD Language	This function is available in LD language	
IL Language	Op1: LD I Terminate ST Q	

9.1.5. RedSwitch

Function	Triggers a redundancy switch	
Inputs	SWITCH: BOOL; Input to trigger the redundancy switch	
Outputs	ERROR: BOOL; TRUE in case of error, e.g. no passive Runtime connected	
Remarks	This function causes a redundancy switch of the Runtime when the input SWITCH becomes TRUE	
ST Language	ERROR := RedSwitch (SWITCH);	
FBD Language	This function is available in FBD language.	
LD Language	This function is available in LD language.	
IL Language	Op1: LD SWITCH RedSwitch ST ERROR	



9.1.6. ApplyRecipeColumn RTK_OnBugCheck

Function Keeps the RTK running after a Blue screen of Death of MS Windows XP.

Inputs Continue : BOOL Set to TRUE if the RTK should remain running after a Blue screen of Death. Set to FALSE to shut down the RTK after a Blue screen of Death considering the Shutdown program if declared.

Outputs Q : BOOL Represents the value of the input.

Remarks Designed for COPA-DATA's RTK Real-time Runtime for Windows XP only.

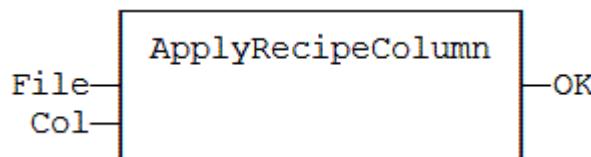


After a Blue screen of Death of MS Windows XP Windows specific functions in the PLC code can not be executed any more. These are: File operations, TCP functions, Serial interface functions, non real time fieldbus drivers, System clock functions, RETAIN variables, writing log messages.

In LD language, the input rung (EN) is the input. The output rung is the output signal Q.

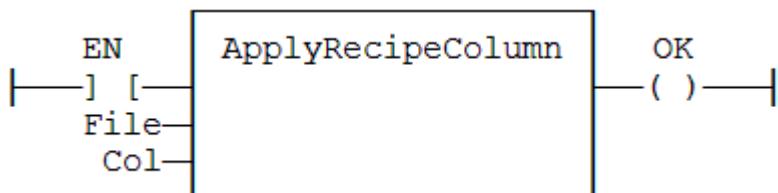
ST Language `Q := RTK_IsBugCheck();`

FBD Language



LD Language

The Function is performed when the EN input is TRUE.



IL Language

`Op1: LD Continue RTK_OnBugCheck ST Q`



9.1.7. Stop

Function	Stops the Runtime	
Inputs	I: BOOL;	Input to stop the Runtime
Outputs	Q: BOOL;	TRUE if successful.
Remarks	This function causes a stop of the Runtime.	
ST Language	Q := Stop(I);	
FBD Language	This function is available in FBD language.	
LD Language	This function is available in LD language.	
IL Language	Op1: LD I Stop ST Q	



9.2. Run Time Options

Run time options can be entered from the **Build** menu of the main window. Run-time options must be defined BEFORE the application is built.

9.2.1. Cycle time

You must specify in this box how the cycles must be triggered at run-time:

- not triggered: the target does not wait between two cycles. The target simply runs as fast as possible.
- triggered: you must specify the duration of a cycle, expressed as a number of micro-seconds. Refer to OEM instructions for explanations of the supported accuracy for the cycle timing of your target system.

9.2.2. Program scheduling

You can define various sampling periods for programs of the application. Default period is 1: the program is executed on each cycle. Giving slower period to some programs is an easy way to give higher priority to some other programs.

To change the scheduling of programs, select them in the left side list and press > or drag them to the grid where you can change for each selected program:

- Its period (how many cycles between two executions of the program).
- Its phase (an offset that enables to dispatch slow programs among few cycles).

The diagram on the top shows the loading of target cycles. On each sample, the diagram shows how many programs are run. You can highlight the scheduling of a program in the diagram by checking it in the grid.

The following Example illustrates the scheduling of an application with three programs. In order to give higher priority to program P3, a period of 2 is given to programs P1 and P2. P2 as an offset 1 so that it is not executed on the same cycle as P1:

Example

	Period	Phase
P1	2	0
P2	2	1
P3	1	0

The diagram below shows which program is executed on each cycle according to this configuration:

P1	P2	P1	P2	P1	P2	P1	P2	P1
P3								
1	2	3	4	5	6	7	8	...

9.2.3. SHLb / SHL_SINT / SHL_USINT / SHL_BYTE

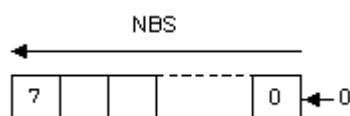
Function Shift bits of a register to the left.

Inputs IN : SINT 8 bit register.

NBS : SINT Number of shifts (each shift is 1 bit).

Outputs Q : SINT Shifted register

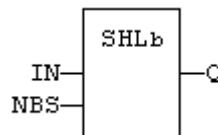
Diagram



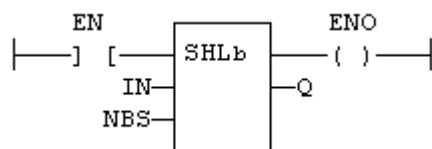
Remarks In LD language, the input rung (EN) enables the operation, and the output rung keeps the state of the input rung. In IL language, the first input must be loaded before the function call. The second input is the operand of the function.

ST Language `Q := SHLb (IN, NBS);`

FBD Language



LD Language The shift is executed only if EN is *TRUE*. ENO has the same value as EN.



IL Language `Op1: LD IN SHLb NBS ST Q`



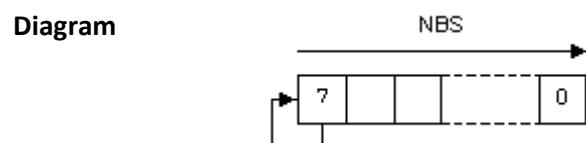
9.2.4. SHLw / SHL_INT / SHL_UINT / SHL_WORD

Function	Shift bits of a register to the left.
Inputs	IN : INT 16 bit register NBS : INT Number of shifts (each shift is 1 bit)
Outputs	Q : INT Shifted register
Diagram	
Remarks	In LD language, the input rung (EN) enables the operation, and the output rung keeps the state of the input rung. In IL language, the first input must be loaded before the function call. The second input is the operand of the function.
ST Language	<code>Q := SHLw (IN, NBS);</code>
FBD Language	
LD Language	The shift is executed only if EN is <i>TRUE</i> . ENO has the same value as EN.
IL Language	<code>Op1: LD IN SHLw NBS ST Q</code>

9.2.5. SHRb / SHR_SINT / SHR_USINT / SHR_BYTE

Function: Shift bits of a register to the right.

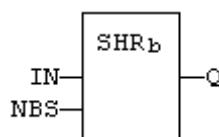
Inputs	IN : SINT	8 bit register
	NBS : SINT	Number of shifts (each shift is 1 bit)
Outputs	Q : SINT	Shifted register



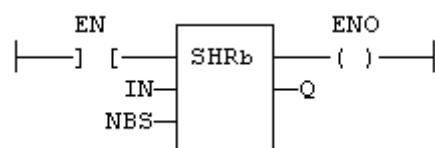
Remarks In LD language, the input rung (EN) enables the operation, and the output rung keeps the state of the input rung. In IL language, the first input must be loaded before the function call. The second input is the operand of the function.

ST Language `Q := SHRb (IN, NBS);`

FBD Language



LD Language The shift is executed only if EN is *TRUE*. ENO has the same value as EN.



IL Language `Op1: LD IN SHRb NBS ST Q`

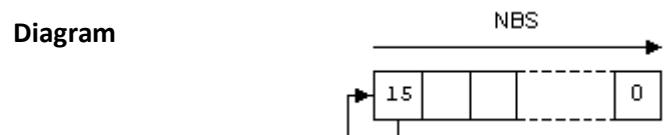
9.2.6. SHRw / SHR_INT / SHR_UINT / SHR_WORD

Function Shift bits of a register to the right.

Inputs IN : INT 16 bit register

NBS : INT Number of shifts (each shift is 1 bit)

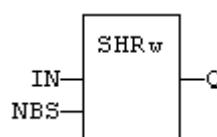
Outputs Q : INT Shifted register



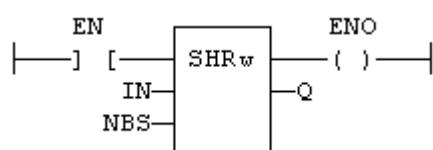
Remarks In LD language, the input rung (EN) enables the operation, and the output rung keeps the state of the input rung. In IL language, the first input must be loaded before the function call. The second input is the operand of the function.

ST Language `Q := SHRw (IN, NBS);`

FBD Language



LD Language The shift is executed only if EN is TRUE. ENO has the same value as EN.



IL Language `Op1: LD IN SHRw NBS ST Q`

9.2.7. SET_DAY_TIME

Function

Sets the system time of Windows operating systems.

Inputs

TM: STRING Input string for date and time. Format: 'DD.MM.YYYY
HH:MM:SS'

SEL: BOOL Sends the value of TM to the operating system with rising edge of this input

Outputs

Q: BOOL TRUE if successful

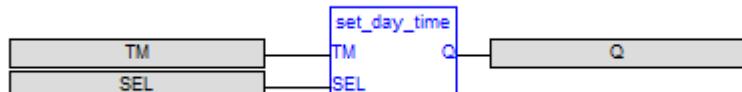
Remarks

Setting the system time with this function can heavily influence the cycle time. Set back the SEL input variable to FALSE right after the function call in order to avoid problems with the operating system.

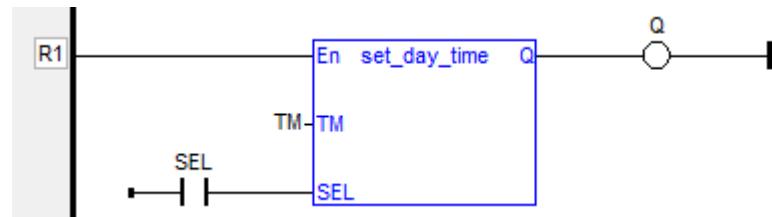
The time on Input TM must be GMT/UTC time. The target system will calculate the local time via its time zone settings.

ST Language

`Q := SET_DAY_TIME (TM, SEL);`

FBD Language**LD Language**

The function is executed only if EN is *TRUE*. The output rung is the result of the function. The output rung is *FALSE* if the EN is *FALSE*.

**IL Language**

`Op1: LD TM SET_DAY_TIME SEL ST Q`